D20

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
H04L 29/06, G07F 7/10, G06F 1/00

A1

(11) International Publication Number: **WO 98/37675**

(43) International Publication Date: 27 August 1998 (27.08.98)

(21) International Application Number: PCT/US98/03236

(22) International Filing Date: 18 February 1998 (18.02.98)

(30) Priority Data:
08/801,026 19 February 1997 (19.02.97) US

(71) Applicant *(for all designated States except US)*: VERIFONE, INC. [US/US]; 4988 Great America Parkway, Santa Clara, CA 95054–1200 (US).

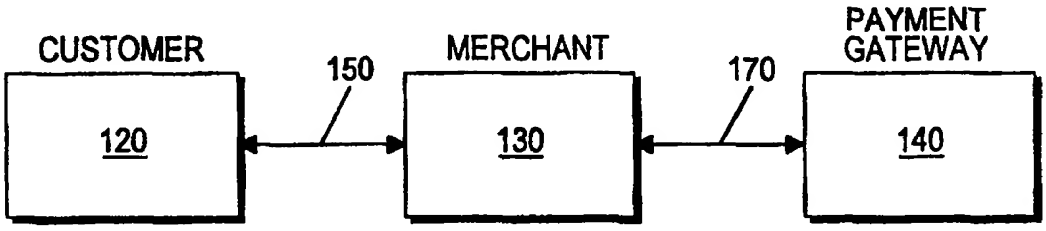
(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: ROWNEY, Kevin, T., B. [US/US]; 748 Duncan Street, San Francisco, CA 94131 (US).

(74) Agents: WARREN, Sanford, E., Jr. et al.; Warren & Perez, Suite 710, 8411 Preston Road, Dallas, TX 75225 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: A SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR SECURE DIGITAL CERTIFICATION OF ELECTRONIC COMMERCE



(57) Abstract

Secure transmission of data is provided between a plurality of computer systems (120, 130, 140) over a public communication system (150, 170), such as the Internet. Secure transmission of data is provided from a party in communication with a first application resident on a first computer (130) which is in communication with a second computer with a certification authority application resident thereon. The second computer (140) is in communication with a third computer utilizing an administrative function resident thereon. The first (130), second and third (140) computers are connected by a network (150, 170), such as the Internet. A name–value pair for certification processing is created on said first computer (130) and transmitted to an administrative function on the third computer (140). Then, the name–value pair is routed to the appropriate certification authority on the second computer. The administrative function also transmits other cerfication information from said admistrative function to said certification authority on the second computer. Until, finally, a certificate is created comprising the name–value pair and the other certification information on the second computer. The certificate is utilized for authenticating indentity of the party.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

A SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR SECURE DIGITAL CERTIFICATION OF ELECTRONIC COMMERCE

Field Of The Invention

The present invention relates to the secure, electronic payment in exchange for goods and services purchased over a communication network, and more specifically, to a system, method and article of manufacture for securely transmitting payment information from a customer to a merchant to a payment gateway and returning a certification, including a credit confidence factor to allow a merchant to determine whether to accept or reject payment information utilizing a flexible, extensible architecture.

The present invention relates to an electronic graphical representation of a monetary system for implementing electronic money payments as an alternative medium of economic exchange to cash, checks, credit and debit cards, and electronic funds transfer. The Electronic-Monetary System is a hybrid of currency, check, card payment systems, and electronic funds transfer systems, possessing many of the benefits of these systems with few of their limitations. The system utilizes electronic representations of money which are designed to be universally accepted and exchanged as economic value by subscribers of the monetary system.

Today, approximately 350 billion coin and currency transactions occur between individuals and institutions every year. The extensive use of coin and currency
transactions has limited the automation of individual transactions such as purchases, fares, and bank account deposits and withdrawals. Individual cash transactions are burdened by the need to have the correct amount of

cash or providing change therefor. Furthermore, the handling and managing of paper cash and coins is inconvenient, costly and time consuming for both individuals and financial institutions.

Although checks may be written for any specific amount up to the amount available in the account, checks have very limited transferability and must be supplied from a physical inventory. Paper-based checking systems do not offer sufficient relief from the limitations of cash transactions, sharing many of the inconveniences of handling currency while adding the inherent delays associated with processing checks. To this end, economic exchange has striven for greater convenience at a lower cost, while also seeking improved security.

Automation has achieved some of these qualities for large transactions through computerized electronic funds transfer ("EFT") systems. Electronic funds transfer is essentially a process of value exchange achieved through the banking system's centralized computer transactions. EFT services are a transfer of payments utilizing electronic "checks," which are used primarily by large commercial organizations.

The Clearing House (ACH) where a user can enter a pre-authorized code and download information with billing occurring later, and a Point Of Sale (POS) system where a transaction is processed by connecting with a central computer for authorization for the transaction granted or denied immediately are examples of EFT systems that are utilized by retail and commercial organizations. However, the payments made through these types of EFT systems are limited in that they cannot be performed without the banking system. Moreover, ACH transactions usually cannot be performed during off business hours.

Home Banking bill payment services are examples of an EFT system used by individuals to make payments from a home computer. Currently, home banking initiatives have found few customers. Of the banks that have offered services for payments, account transfers and information over the telephone lines using personal computers, less than one percent of the bank's customers are using the service. One reason that Home Banking has not been a successful product is because the customer cannot deposit and withdraw money as needed in this type of system.

Current EFT systems, credit cards, or debit cards, which are used in conjunction with an on-line system to transfer money between accounts, such as between the account of a merchant and that of a customer, cannot satisfy the need for an automated transaction system providing an ergonomic interface. Examples of EFT systems which provide non-ergonomic interfaces are disclosed in US Patents Numbers 5,476,259; 5,459,304; 5,452,352; 5,448,045; 5,478,993; 5,455,407; 5,453,601; 5,465,291; and 5,485,510.

To implement an automated, convenient transaction that can dispense some form of economic value, there has been a trend towards off-line payments. For example, numerous ideas have been proposed for some form of "electronic money" that can be used in cashless payment transactions as alternatives to the traditional currency and check types of payment systems. See U.S. Pat. No. 4,977,595, entitled "METHOD AND APPARATUS FOR IMPLEMENTING ELECTRONIC CASH," and U.S. Pat. No. 4,305,059, entitled "MODULAR FUNDS TRANSFER SYSTEM."

The more well known techniques include magnetic stripe cards purchased for a given amount and from which a prepaid value can be deducted for specific purposes. Upon exhaustion of the economic value, the cards are thrown away. Other examples include memory cards or so called smart cards which are capable of repetitively storing information representing value that is likewise deducted for specific purposes.

It is desirable for a computer operated under the control of a merchant to obtain information offered by a customer and transmitted by a computer operating under the control of the customer over a publicly accessible packet-switched network (e.g., the Internet) to the computer operating under the control of the merchant, without risking the exposure of the information to interception by third parties that have access to the network, and to assure that the information is from an authentic source. It is further desirable for the merchant to transmit information, including a subset of the information provided by the customer, over such a network to a payment gateway computer system that is authorized, by a bank or other financial institution that has the responsibility of providing payment on behalf of the customer, to authorize a commercial transaction on behalf of such a financial institution, without the risk of exposing that information to interception by third parties. Such institutions include, for example, financial institutions offering credit or debit card services.

One such attempt to provide such a secure transmission channel is a secure payment technology such as Secure Electronic Transaction (hereinafter "SET"), jointly developed by the Visa and MasterCard card associations, and described in Visa and MasterCard's *Secure Electronic Transaction (SET) Specification*, February 23, 1996, hereby incorporated by reference. Other such secure payment technologies include Secure Transaction Technology

("STT"), Secure Electronic Payments Protocol ("SEPP"), Internet Keyed Payments ("iKP"), Net Trust, and Cybercash Credit Payment Protocol. One of ordinary skill in the art readily comprehends that any of the secure payment technologies can be substituted for the SET protocol without undue experimentation. Such secure payment technologies require the customer to operate software that is compliant with the secure payment technology, interacting with third-party certification authorities, thereby allowing the customer to transmit encoded information to a merchant, some of which may be decoded by the merchant, and some which can be decoded only by a payment gateway specified by the customer.

Another such attempt to provide such a secure transmission channel is a general-purpose secure communication protocol such as Netscape, Inc.'s Secure Sockets Layer (hereinafter "SSL") , as described in Freier, Karlton & Kocher (hereinafter "Freier"), *The SSL Protocol Version 3.0*, March 1996, and hereby incorporated by reference. SSL provides a means for secure transmission between two computers. SSL has the advantage that it does not require special-purpose software to be installed on the customer's computer because it is already incorporated into widely available software that many people utilize as their standard Internet access medium, and does not require that the customer interact with any third-party certification authority. Instead, the support for SSL may be incorporated into software already in use by the customer, e.g., the Netscape Navigator World Wide Web browsing tool. However, although a computer on an SSL connection may initiate a second SSL connection to another computer, a drawback to the SSL approach is each SSL connection supports only a two-computer connection. Therefore, SSL does not provide a mechanism for transmitting encoded information to a merchant for retransmission to a payment gateway such that a subset of the information is readable to the payment gateway

but not to the merchant. Although SSL allows for robustly secure two-party data transmission, it does not meet the ultimate need of the electronic commerce market for robustly secure three-party data transmission. Other examples of general-purpose secure communication protocols include Private Communications Technology ("PCT") from Microsoft, Inc., Secure Hyper-Text Transport Protocol ("SHTTP") from Theresa Systems, Shen, Kerberos, Photuris, Pretty Good Privacy ("PGP") and Ipv6 which meets the IPSEC criteria. One of ordinary skill in the art readily comprehends that any of the general-purpose secure communication protocols can be substituted for the SSL transmission protocol without undue experimentation.

Banks desire an internet payment solution that emulates existing Point of Sale (POS) applications that are currently installed on their host computers, and require minimal changes to their host systems. This is a critical requirement since any downtime for a banks host computer system represents an enormous expense. Currently, Verifone supports over fourteen hundred different payment-related applications. The large number of applications is necessary to accommodate a wide variety of host message formats, diverse methods for communicating to a variety of hosts with different dial-up and direct-connect schemes, and different certification around the world. In addition, there are a wide variety of business processes that dictate how a Point of Sale (POS) terminal queries a user for data and subsequently displays the data. Also, various vertical market segments, such as hotels, car rental agencies, restaurants, retail sales, mail sales / telephone sales require interfaces for different types of data to be entered, and provide different discount rates to merchants for complying with various data types. Moreover, a plethora of report generation mechanisms and formats are utilized by merchants that banking organizations work with.

Banks are unwilling to converge on "standards" since convergence would facilitate switching from one acquiring bank to another by merchants. In general, banks desire to increase the cost that a merchant incurs in switching from one acquiring bank to another acquiring bank. This is accomplished by supplying a merchant with a terminal that only communicates utilizing the bank's proprietary protocol, and by providing other value-added services that a merchant may not be able to obtain at another bank.

Internet-based payment solutions require additional security measures that are not found in conventional POS terminals. This additional requirement is necessitated because internet communication is done over publicly-accessible, unsecured communication line in stark contrast to the private, secure, dedicated phone or leased line service utilized between a traditional merchant and an acquiring bank. Thus, it is critical that any solution utilizing the internet for a communication backbone, employ some form of cryptography.

As discussed above, the current state-of-the-art in internet based payment processing is a protocol referred to as SET. Since the SET messages are uniform across all implementations, banks cannot differentiate themselves in any reasonable way. Also, since SET is not a proper superset of all protocols utilized today, there are bank protocols which cannot be mapped or translated into SET because they require data elements for which SET has no placeholder. Further, SET only handles the message types directly related to authorizing and capturing credit card transactions and adjustments to these authorizations or captures. In a typical POS terminal in the physical world, these messages comprise almost the entire volume of

the total number of messages between the merchant and the authorizing bank, but only half of the total number of different message types. These message types, which are used infrequently, but which are critical to the operation of the POS terminal must be supported for proper transaction processing.

SUMMARY OF THE INVENTION

According to a broad aspect of a preferred embodiment of the invention, secure transmission of data is provided between a plurality of computer systems over a public communication system, such as the Internet. Secure transmission of data is provided from a party in communication with a first application resident on a first computer which is in communication with a second computer with a certification authority application resident thereon. The second computer is in communication with a third computer utilizing an administrative function resident thereon. The first, second and third computers are connected by a network, such as the Internet. A name-value pair for certification processing is created on said first computer and transmitted to an administrative function on the third computer. Then, the name-value pair is routed to the appropriate certification authority on the second computer. The administrative function also transmits other certification information from said administrative function to said certification authority on the second computer. Until, finally, a certificate is created comprising the name-value pair and the other certification information on the second computer.

DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, aspects and advantages are better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

Figure **1A** is a block diagram of a representative hardware environment in accordance with a preferred embodiment;

Figure **1B** depicts an overview in accordance with a preferred embodiment;

Figure **1C** is a block diagram of the system in accordance with a preferred embodiment;

Figure **2** depicts a more detailed view of a customer computer system in communication with merchant system under the Secure Sockets Layer protocol in accordance with a preferred embodiment;

Figure **3** depicts an overview of the method of securely supplying payment information to a payment gateway in order to obtain payment authorization in accordance with a preferred embodiment;

Figure **4** depicts the detailed steps of generating and transmitting a payment authorization request in accordance with a preferred embodiment;

Figures **5A** through **5F** depict views of the payment authorization request and its component parts in accordance with a preferred embodiment;

Figures **6A** and **6B** depict the detailed steps of processing a payment authorization request and generating and transmitting a payment authorization request response in accordance with a preferred embodiment;

Figures **7A** through **7J** depict views of the payment authorization response and its component parts in accordance with a preferred embodiment;

Figure **8** depicts the detailed steps of processing a payment authorization response in accordance with a preferred embodiment;

Figure **9** depicts an overview of the method of securely supplying payment capture information to a payment gateway in accordance with a preferred embodiment;

Figure **10** depicts the detailed steps of generating and transmitting a payment capture request in accordance with a preferred embodiment;

Figures **11A** through **11F** depict views of the payment capture request and its component parts in accordance with a preferred embodiment;

Figures **12A** and **12B** depict the detailed steps of processing a payment capture request and generating and transmitting a payment capture request response in accordance with a preferred embodiment;

Figures **13A** through **13F** depict views of the payment capture response and its component parts in accordance with a preferred embodiment;

Figure **14** depicts the detailed steps of processing a payment capture response in accordance with a preferred embodiment;

Figure **15A** & **15B** depicts transaction processing of merchant and consumer transactions in accordance with a preferred embodiment;

Figure **16** illustrates a transaction class hierarchy block diagram in accordance with a preferred embodiment;

Figure **17** shows a typical message flow between the consumer, merchant, VPOS terminal and the Gateway in accordance with a preferred embodiment;

Figures **18A-E** are block diagrams of the extended SET architecture in accordance with a preferred embodiment;

Figure **19** is a flowchart of VPOS merchant pay customization in accordance with a preferred embodiment;

Figures **20A-20H** are block diagrams and flowcharts setting forth the detailed logic of thread processing in accordance with a preferred embodiment;

Figure **21** is a detailed diagram of a multithreaded gateway engine in accordance with a preferred embodiment;

Figure **22** is a flow diagram in accordance with a preferred embodiment;

Figure **23** illustrates a Gateway's role in a network in accordance with a preferred embodiment;

Figure **24** is a block diagram of the Gateway in accordance with a preferred embodiment;

Figure **25** is a block diagram of the vPOS Terminal Architecture in accordance with a preferred embodiment;

Figure **26** is an architecture block diagram in accordance with a preferred embodiment;

Figure **27** is a block diagram of the payment manager architecture in accordance with a preferred embodiment;

Figure **28** is a Consumer Payment Message Sequence Diagram in accordance with a preferred embodiment of the invention;

Figure **29** is an illustration of a certificate issuance form in accordance with a preferred embodiment;

Figure **30** illustrates a certificate issuance response in accordance with a preferred embodiment;

Figure **31** illustrates a collection of payment instrument holders in accordance with a preferred embodiment;

Figure **32** illustrates the default payment instrument bitmap in accordance with a preferred embodiment;

Figure **33** illustrates a selected payment instrument with a fill in the blanks for the cardholder in accordance with a preferred embodiment;

Figure **34** illustrates a coffee purchase utilizing the newly defined VISA card in accordance with a preferred embodiment of the invention;

Figure **35** is a flowchart of conditional authorization of payment in accordance with a preferred embodiment; and

Figures **36-48** are screen displays in accordance with a preferred embodiment.

DETAILED DESCRIPTION

A preferred embodiment of a system in accordance with the present invention is preferably practiced in the context of a personal computer such as the IBM PS/2, Apple Macintosh computer or UNIX based workstation. A representative hardware environment is depicted in Figure **1A**, which illustrates a typical hardware configuration of a workstation in accordance with a preferred embodiment having a central processing unit **10**, such as a microprocessor, and a number of other units interconnected via a system bus **12**. The workstation shown in Figure **1A** includes a Random Access Memory (RAM) **14**, Read Only Memory (ROM) **16**, an I/O adapter **18** for connecting peripheral devices such as disk storage units **20** to the bus **12**, a user interface adapter **22** for connecting a keyboard **24**, a mouse **26**, a speaker **28**, a microphone **32**, and/or other user interface devices such as a touch screen (not shown) to the bus **12**, communication adapter **34** for connecting the workstation to a communication network (e.g., a data processing network) and a display adapter **36** for connecting the bus **12** to a display device **38**. The workstation typically has resident thereon an operating system such as the Microsoft Windows Operating System (OS), the IBM OS/2 operating system, the MAC OS, or UNIX operating system. Those skilled in the art appreciates that the present invention may also be implemented on platforms and operating systems other than those mentioned.

A preferred embodiment is written using JAVA, C, and the C++ language and utilizes object oriented programming methodology. Object oriented programming (OOP) has become increasingly used to develop complex applications. As OOP moves toward the mainstream of software design and

development, various software solutions require adaptation to make use of the benefits of OOP. A need exists for these principles of OOP to be applied to a messaging interface of an electronic messaging system such that a set of OOP classes and objects for the messaging interface can be provided.

OOP is a process of developing computer software using objects, including the steps of analyzing the problem, designing the system, and constructing the program. An object is a software package that contains both data and a collection of related structures and procedures. Since it contains both data and a collection of structures and procedures, it can be visualized as a self-sufficient component that does not require other additional structures, procedures or data to perform its specific task. OOP, therefore, views a computer program as a collection of largely autonomous components, called objects, each of which is responsible for a specific task. This concept of packaging data, structures, and procedures together in one component or module is called encapsulation.

In general, OOP components are reusable software modules which present an interface that conforms to an object model and which are accessed at run-time through a component integration architecture. A component integration architecture is a set of architecture mechanisms which allow software modules in different process spaces to utilize each others capabilities or functions. This is generally done by assuming a common component object model on which to build the architecture.

It is worthwhile to differentiate between an object and a class of objects at this point. An object is a single instance of the class of objects, which is often just called a class. A class of objects can be viewed as a blueprint, from which many objects can be formed.

OOP allows the programmer to create an object that is a part of another object. For example, the object representing a piston engine is said to have a composition-relationship with the object representing a piston. In reality, a piston engine comprises a piston, valves and many other components; the fact that a piston is an element of a piston engine can be logically and semantically represented in OOP by two objects.

OOP also allows creation of an object that "depends from" another object. If there are two objects, one representing a piston engine and the other representing a piston engine wherein the piston is made of ceramic, then the relationship between the two objects is not that of composition. A ceramic piston engine does not make up a piston engine. Rather it is merely one kind of piston engine that has one more limitation than the piston engine; its piston is made of ceramic. In this case, the object representing the ceramic piston engine is called a derived object, and it inherits all of the aspects of the object representing the piston engine and adds further limitation or detail to it. The object representing the ceramic piston engine "depends from" the object representing the piston engine. The relationship between these objects is called inheritance.

When the object or class representing the ceramic piston engine inherits all of the aspects of the objects representing the piston engine, it inherits the thermal characteristics of a standard piston defined in the piston engine class. However, the ceramic piston engine object overrides these ceramic specific thermal characteristics, which are typically different from those associated with a metal piston. It skips over the original and uses new functions related to ceramic pistons. Different kinds of piston engines have different characteristics, but may have the same underlying functions

associated with it (e.g., how many pistons in the engine, ignition sequences, lubrication, etc.). To access each of these functions in any piston engine object, a programmer would call the same functions with the same names, but each type of piston engine may have different/overriding implementations of functions behind the same name. This ability to hide different implementations of a function behind the same name is called polymorphism and it greatly simplifies communication among objects.

With the concepts of composition-relationship, encapsulation, inheritance and polymorphism, an object can represent just about anything in the real world. In fact, our logical perception of the reality is the only limit on determining the kinds of things that can become objects in object-oriented software. Some typical categories are as follows:

- Objects can represent physical objects, such as automobiles in a traffic-flow simulation, electrical components in a circuit-design program, countries in an economics model, or aircraft in an air-traffic-control system.
- Objects can represent elements of the computer-user environment such as windows, menus or graphics objects.
- An object can represent an inventory, such as a personnel file or a table of the latitudes and longitudes of cities.
- An object can represent user-defined data types such as time, angles, and complex numbers, or points on the plane.

With this enormous capability of an object to represent just about any logically separable matters, OOP allows the software developer to design and implement a computer program that is a model of some aspects of reality, whether that reality is a physical entity, a process, a system, or a composition of matter. Since the object can represent anything, the

software developer can create an object which can be used as a component in a larger software project in the future.

If 90% of a new OOP software program consists of proven, existing components made from preexisting reusable objects, then only the remaining 10% of the new software project has to be written and tested from scratch. Since 90% already came from an inventory of extensively tested reusable objects, the potential domain from which an error could originate is 10% of the program. As a result, OOP enables software developers to build objects out of other, previously built, objects.

This process closely resembles complex machinery being built out of assemblies and sub-assemblies. OOP technology, therefore, makes software engineering more like hardware engineering in that software is built from existing components, which are available to the developer as objects. All this adds up to an improved quality of the software as well as an increased speed of its development.

Programming languages are beginning to fully support the OOP principles, such as encapsulation, inheritance, polymorphism, and composition-relationship. With the advent of the C++ language, many commercial software developers have embraced OOP. C++ is an OOP language that offers a fast, machine-executable code. Furthermore, C++ is suitable for both commercial-application and systems-programming projects. For now, C++ appears to be the most popular choice among many OOP programmers, but there is a host of other OOP languages, such as Smalltalk, common lisp object system (CLOS), and Eiffel. Additionally, OOP capabilities are being added to more traditional popular computer programming languages such as Pascal.

The benefits of object classes can be summarized, as follows:

- Σ *Objects* and their corresponding classes break down complex programming problems into many smaller, simpler problems.
- Σ *Encapsulation* enforces data abstraction through the organization of data into small, independent objects that can communicate with each other. Encapsulation protects the data in an object from accidental damage, but allows other objects to interact with that data by calling the object's member functions and structures.
- Σ *Subclassing* and inheritance make it possible to extend and modify objects through deriving new kinds of objects from the standard classes available in the system. Thus, new capabilities are created without having to start from scratch.
- Σ *Polymorphism* and multiple inheritance make it possible for different programmers to mix and match characteristics of many different classes and create specialized objects that can still work with related objects in predictable ways.
- Σ *Class hierarchies* and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.
- Σ *Libraries* of reusable classes are useful in many situations, but they also have some limitations. For example:
- Σ *Complexity.* In a complex system, the class hierarchies for related classes can become extremely confusing, with many dozens or even hundreds of classes.
- Σ *Flow of control.* A program written with the aid of class libraries is still responsible for the flow of control (i.e., it must control the interactions among all the objects created from a particular library). The

programmer has to decide which functions to call at what times for which kinds of objects.

- *Duplication of effort.* Although class libraries allow programmers to use and reuse many small pieces of code, each programmer puts those pieces together in a different way. Two different programmers can use the same set of class libraries to write two programs that do exactly the same thing but whose internal structure (i.e., design) may be quite different, depending on hundreds of small decisions each programmer makes along the way. Inevitably, similar pieces of code end up doing similar things in slightly different ways and do not work as well together as they should.

Class libraries are very flexible. As programs grow more complex, more programmers are forced to reinvent basic solutions to basic problems over and over again. A relatively new extension of the class library concept is to have a framework of class libraries. This framework is more complex and consists of significant collections of collaborating classes that capture both the small scale patterns and major mechanisms that implement the common requirements and design in a specific application domain. They were first developed to free application programmers from the chores involved in displaying menus, windows, dialog boxes, and other standard user interface elements for personal computers.

Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the flow of control. This was appropriate for

printing out paychecks, calculating a mathematical table, or solving other problems with a program that executed in just one way.

The development of graphical user interfaces began to turn this procedural programming arrangement inside out. These interfaces allow the user, rather than program logic, to drive the program and decide when certain actions should be performed. Today, most personal computer software accomplishes this by means of an event loop which monitors the mouse, keyboard, and other sources of external events and calls the appropriate parts of the programmer's code according to actions that the user performs. The programmer no longer determines the order in which events occur. Instead, a program is divided into separate pieces that are called at unpredictable times and in an unpredictable order. By relinquishing control in this way to users, the developer creates a program that is much easier to use. Nevertheless, individual pieces of the program written by the developer still call libraries provided by the operating system to accomplish certain tasks, and the programmer must still determine the flow of control within each piece after it's called by the event loop. Application code still "sits on top of" the system.

Even event loop programs require programmers to write a lot of code that should not need to be written separately for every application. The concept of an application framework carries the event loop concept further. Instead of dealing with all the nuts and bolts of constructing basic menus, windows, and dialog boxes and then making these things all work together, programmers using application frameworks start with working application code and basic user interface elements in place. Subsequently, they build from there by replacing some of the generic capabilities of the framework with the specific capabilities of the intended application.

Application frameworks reduce the total amount of code that a programmer has to write from scratch. However, because the framework is really a generic application that displays windows, supports copy and paste, and so on, the programmer can also relinquish control to a greater degree than event loop programs permit. The framework code takes care of almost all event handling and flow of control, and the programmer's code is called only when the framework needs it (e.g., to create or manipulate a proprietary data structure).

A programmer writing a framework program not only relinquishes control to the user (as is also true for event loop programs), but also relinquishes the detailed flow of control within the program to the framework. This approach allows the creation of more complex systems that work together in interesting ways, as opposed to isolated programs, having custom code, being created over and over again for similar problems.

Thus, as is explained above, a framework basically is a collection of cooperating classes that make up a reusable design solution for a given problem domain. It typically includes objects that provide default behavior (e.g., for menus and windows), and programmers use it by inheriting some of that default behavior and overriding other behavior so that the framework calls application code at the appropriate times.

There are three main differences between frameworks and class libraries:

- *Behavior versus protocol.* Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a

programmer is supposed to provide versus what the framework provides.

- *Call versus override.* With a class library, the code the programmer instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together.
- *Implementation versus design.* With class libraries, programmers reuse only implementations, whereas with frameworks, they reuse design. A framework embodies the way a family of related programs or pieces of software work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Thus, through the development of frameworks for solutions to various problems and programming tasks, significant reductions in the design and development effort for software can be achieved. A preferred embodiment of the invention utilizes HyperText Markup Language (HTML) to implement documents on the Internet together with a general-purpose secure communication protocol for a transport medium between the client and the merchant. HTTP or other protocols could be readily substituted for HTML without undue experimentation. Information on these products is available

in T. Berners-Lee, D. Connoly, "RFC 1866: Hypertext Markup Language - 2.0" (Nov. 1995); and R. Fielding, H, Frystyk, T. Berners-Lee, J. Gettys and J.C. Mogul, "Hypertext Transfer Protocol -- HTTP/1.1: HTTP Working Group Internet Draft" (May 2, 1996). HTML is a simple data format used to create hypertext documents that are portable from one platform to another. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML has been in use by the World-Wide Web global information initiative since 1990. HTML is an application of ISO Standard 8879:1986 Information Processing Text and Office Systems; Standard Generalized Markup Language (SGML).

To date, Web development tools have been limited in their ability to create dynamic Web applications which span from client to server and interoperate with existing computing resources. Until recently, HTML has been the dominant technology used in development of Web-based solutions. However, HTML has proven to be inadequate in the following areas:

- o Poor performance;
- o Restricted user interface capabilities;
- o Can only produce static Web pages;
- o Lack of interoperability with existing applications and data; and
- o Inability to scale.

Sun Microsystem's Java language solves many of the client-side problems by:

- o Improving performance on the client side;
- o Enabling the creation of dynamic, real-time Web applications; and
- o Providing the ability to create a wide variety of user interface components.

With Java, developers can create robust User Interface (UI) components. Custom "widgets" (e.g. real-time stock tickers, animated icons, etc.) can be created, and client-side performance is improved. Unlike HTML, Java supports the notion of client-side validation, offloading appropriate processing onto the client for improved performance. Dynamic, real-time Web pages can be created. Using the above-mentioned custom UI components, dynamic Web pages can also be created.

Sun's Java language has emerged as an industry-recognized language for "programming the Internet." Sun defines Java as: "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language. Java supports programming for the Internet in the form of platform-independent Java applets." Java applets are small, specialized applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g. simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible browser (e.g. Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on C++. Sun's Java literature states that Java is basically "C++, with extensions from Objective C for more dynamic method resolution".

Another technology that provides similar function to JAVA is provided by Microsoft and ActiveX Technologies, to give developers and Web designers wherewithal to build dynamic content for the Internet and personal computers. ActiveX includes tools for developing animation, 3-D virtual reality, video and other multimedia content. The tools use Internet

standards, work on multiple platforms, and are being supported by over 100 companies. The group's building blocks are called ActiveX Controls, small, fast components that enable developers to embed parts of software in hypertext markup language (HTML) pages. ActiveX Controls work with a variety of programming languages including Microsoft Visual C++, Borland Delphi, Microsoft Visual Basic programming system and, in the future, Microsoft's development tool for Java, code named "Jakarta." ActiveX Technologies also includes ActiveX Server Framework, allowing developers to create server applications. One of ordinary skill in the art readily recognizes that ActiveX could be substituted for JAVA without undue experimentation to practice the invention.

Figure **1B** depicts an overview of the present invention. Customer computer system **120** is in communication with merchant computer system **130**. The customer-merchant session **150** operates under a general-purpose secure communication protocol such as the SSL protocol. Merchant computer system **130** is additionally in communication with payment gateway computer system **140**. A payment gateway is a system that provides electronic commerce services in support of a bank or other financial institution, and that interfaces to the financial institution to support the authorization and capture of transactions. The customer-institution session **170** operates under a variant of a secure payment technology such as the SET protocol, as described herein, referred to as Merchant-Originated Secure Electronic Transactions ("MOSET"), as is more fully described herein.

Customer-to-Merchant Communication

Figure **2** depicts a more detailed view of customer computer system **120** in communication with merchant system **130** using customer-merchant

session **150** operating under the SSL protocol as documented in Freier and incorporated by reference.

Customer computer system **120** initiates communication with merchant computer system **130** using any well-known access protocol, e.g., Transmission Control Protocol/Internet Protocol ("TCP/IP"). A description of TCP/IP is provided in Information Sciences Institute, "Transmission Control Protocol DARPA Internet Program Protocol Specification (RFC 793)" (September, 1981), and Information Sciences Institute, "Internet Protocol DARPA Internet Program Protocol Specification (RFC 791)" (September, 1981). In this implementation, customer computer system **120** acts as a client and merchant computer system **130** acts as a server.

Customer computer system **120** initiates communication by sending "client hello" message **210** to the merchant computer system **130**. When a client first connects to a server it is required to send the client hello message **210** as its first message. The client can also send a client hello message **210** in response to a hello request on its own initiative in order to renegotiate the security parameters in an existing connection. The client hello message includes a random structure, which is used later in the protocol. Specifically, the random structure includes the current time and date in standard UNIX 32-bit format according to the sender's internal clock and twenty-eight bytes of data generated by a secure random number generator. The client hello message **210** further includes a variable length session identifier. If not empty, the session identifier value identifies a session between the same client and server whose security parameters the client wishes to reuse. The session identifier may be from an earlier connection, the current connection, or another currently active connection. It is useful to specify the current connection if the client only wishes to update the

random structures and derived values of a connection. It is useful to specify another currently active connection if the client wishes to establish several simultaneous independent secure connections to the same server without repeating the full-handshake protocol. Client hello message **210** further includes an indicator of the cryptographic algorithms supported by the client in order of the client's preference, ordered according to client preference.

In response to client hello message **210**, if merchant computer system **130** wishes to correspond with customer computer system **120**, it responds with server hello message **215**. If merchant computer system **130** does not wish to communicate with customer computer system **120**, it responds with a message, not shown, indicating refusal to communicate.

Server hello message **215** includes a random structure, which is used later in the protocol. The random structure in server hello message **215** is in the same format as, but has contents independent of, the random structure in client hello message **210**. Specifically, the random structure includes the current time and date in standard UNIX 32-bit format according to the sender's internal clock and twenty-eight bytes of data generated by a secure random number generator. Server hello message **215** further includes a variable length session identifier. The session identifier value identifies a new or existing session between the same client and server. Server hello message **215** further includes an indicator of the cryptographic algorithms selected from among the algorithms specified by client hello message **210**, which is utilized in further encrypted communications.

Optionally, Merchant computer system **130** transmits a server certificate **220**. If transmitted, server certificate **130** enables customer computer system **120** to authenticate the identity of merchant computer system **130**.

If merchant computer system **130** does not transmit a server certificate **220**, or if server certificate **220** is suitable only for authentication, it may optionally transmit a server key exchange message **225**. Server key exchange message **225** identifies a key that may be used by customer computer system **120** to decrypt further messages sent by merchant computer system **130**.

After transmitting server hello message **215**, and optionally transmitting server certificate **220** or server key exchange message **225**, merchant computer system **130** transmits a server hello done message **230** and waits for a further response from customer computer system **120**.

Customer computer system **120** optionally transmits client certificate **240** to merchant computer system **130**. If transmitted, client certificate **240** enables merchant computer system **130** to authenticate the identity of customer computer system **120**. Alternatively, customer computer system **120** may transmit a no-client-certificate alert **245**, to indicate that the customer has not registered with any certification authority.

If customer computer system **130** does not transmit a client certificate **240**, or if client certificate **240** is suitable only for authentication, customer computer system **130** may optionally transmit a client key exchange message **250**. Client key exchange message **250** identifies a key that may be used by merchant computer system **130** to decrypt further messages sent by customer computer system **120**.

After optionally transmitting client certificate **240**, no-client-certificate alert **245**, and/or client key exchange message **250**, customer computer system **120** transmits a finished message **260**.

At this point, customer computer system **120** and merchant computer system **130** have:

1) negotiated an encryption scheme that may be commonly employed in further communications, and
2) have communicated to each other a set of encryption keys that may be used to decrypt further communications between the two computer systems.

Customer computer system **120** and merchant computer system **130** may thereafter engage in secure communications **270** with less risk of interception by third parties.

Among the messages communicated by customer computer system **120** to merchant computer system **130** may be messages that specify goods or services to be ordered and payment information, such as a credit card number and related information, collectively referred to as "payment information," that may be used to pay for the goods and/or services ordered. In order to obtain payment, the merchant must supply this information to the bank or other payment gateway responsible for the proffered payment method. This enables the merchant to perform payment authorization and payment capture. Payment authorization is the process by which permission is granted by a payment gateway operating on behalf of a financial institution to authorize payment on behalf of the financial institution. This is a process that assesses transaction risk, confirms that a

given transaction does not raise the account holder's debt above the account's credit limit, and reserves the specified amount of credit. Payment capture is the process that triggers the movement of funds from the financial institution to the merchant's account.

Payment Authorization

Merchants utilize point-of-sale products for credit and debit transactions on a daily basis. An embodiment in accordance with the subject invention allows an acquirer processor to accept transactions from internet storefronts without altering a current host environment.

The system easily converts payment protocol messages and simultaneously manages transactions from a number of internet merchant servers. As the number of transactions grows, the payment gateway can be scaled to handle the increased business, and it can be configured to work with specific business processes used by the acquirer/processor. Thus, the payment gateway supports internet processing utilizing payment processing operations.

The payment gateway provides support for configuring and installing the internet payment capability utilizing existing host point-of-sale technology. The payment gateway also provides an intuitive Graphical User Interface (GUI) with support built in to accommodate future payment instruments such as debit cards, electronic checks, electronic cash and micropayments. The payment gateway implements secure transactions using RSA public-key cryptography and the MasterCard/Visa Secure Electronic Transaction (SET) protocol. The gateway also provides full functionality for merchant payment processing including authorization, capture, settlement and reconciliation while providing monitor activity with reporting and tracking of transactions sent over the internet. Finally, the payment gateway also implements internet payment procedures that match current processor business models

to ensure consistency for merchants. Handling internet transactions is destined to become a necessary function for every payment procrocessing system. Today, merchants oftent transmit data inefficiently. Some fax the information or waste time keying data into a non-internet system.

Figure **3** depicts an overview of the method of securely supplying payment information to a payment gateway in order to obtain payment authorization. In function block **310**, merchant computer system **130** generates a payment authorization request **315** and transmits it to payment gateway computer system **140**. In function block **330**, payment gateway system **140** processes the payment authorization request, generates a payment authorization response **325** and transmits it to merchant computer system **130**. In function block **320**, merchant computer system **130** processes payment authorization response **325** and determines whether payment for the goods or services sought to be obtained by the customer has been authorized.

Payment Authorization Request Generation

Figure **4** depicts the detailed steps of generating and transmitting a payment authorization request. Figures **5A** through **5F** depict views of the payment authorization request and its component parts. In function block **410**, merchant computer system **130** creates a basic authorization request **510**. The basic authorization request is a data area that includes all the information for determining whether a request should be granted or denied. Specifically, it includes such information as the party who is being charged, the amount to be charged, the account number of the account to be charged, and any additional data, such as passwords, needed to validate the charge. This information is either calculated based upon prior customer merchandise selection, or provided by the customer over the secure link **270** established in the customer-merchant general-purpose secure

communication protocol session. Fig **5A** depicts a basic authorization request **510**.

In function block **420**, merchant computer system **130** combines basic authorization request **510**, a copy of its encryption public key certificate **515** and a copy of its signature public key certificate **520**. Merchant computer system **130** calculates a digital signature **525** for the combined contents of the combined block **530** comprising basic authorization request **510**, the encryption public key certificate **515** and the signature public key certificate **520**, and appends it to the combination of the combined basic authorization request **510**, the encryption public key certificate **515** and the signature public key certificate **520**. The merchant computer system calculates digital signature **525** by first calculating a "message digest" based upon the contents of the combined basic authorization request **510**, the encryption public key certificate **515** and the signature public key certificate **520**. A message digest is the fixed-length result that is generated when a variable length message is fed into a one-way hashing function. Message digests help verify that a message has not been altered because altering the message would change the digest. The message digest is then encrypted using the merchant computer system's **130** digital signature private key, thus forming a digital signature.

Figure **5B** depicts the combined block **530** formed by function block **420** and containing basic authorization request **510**, the encryption public key certificate **515**, the signature public key certificate **520**, and digital signature **525**.

In function block **430**, merchant computer system **130** generates a random encryption key RK-0 **540**, denoted as RK-0. Random encryption key RK-0

540 is a symmetric encryption key. A symmetric encryption key is a key characterized by the property that a message encrypted with a symmetric key can be decrypted with that same key. This is contrasted with an asymmetric key pair, such as a public-key/private-key key pair, where a message encrypted with one key of the key pair may only be decrypted with the other key of the same key pair. Figure **5C** depicts random encryption key RK-0 **540**.

In function block **440**, merchant computer system **130** encrypts combined block **530** using random encryption key RK-0 **540** to form encrypted combined block **550**. Figure **5D** depicts encrypted combined block **550**. The encryption state of encrypted combined block **550** is graphically shown by random key lock **555**, which indicates that encrypted combined block **550** is encrypted using random key RK-0 **540**.

In function block **450**, merchant computer system **130** encrypts random encryption key RK-0 **540** using the public key of payment gateway system **140** to form encrypted random key **560**. Figure **5E** depicts encrypted random key **560**. The encryption state of encrypted random key **560** is graphically shown by payment gateway public key lock **565**, which indicates that encrypted random key **560** is encrypted using the payment gateway public key.

In function block **460**, merchant computer system **130** concatenates encrypted combined block **550** and encrypted random key 560 to form merchant authorization request **315**. Figure **5F** depicts merchant authorization request **315** comprising encrypted combined block **550** and encrypted random key **560**. In function block **470**, merchant computer

system **130** transmits merchant authorization request **315** to payment gateway system **140**.

Payment Authorization Request Processing

Figure **6** depicts the detailed steps of processing a payment authorization request and generating and transmitting a payment authorization request response. Function blocks **610** through **630** depict the steps of processing a payment authorization request, while function blocks **635** through **685** depict the steps of generating and transmitting a payment authorization request response.

In function block **610**, payment gateway computer system **140** applies its private key to encrypted random key **560** contained within received merchant authorization request **315**, thereby decrypting it and obtaining a cleartext version of random key RK-0 **540**. In function block **615**, payment gateway computer system **140** applies random key RK-0 **540** to encrypted combined block **550**, thereby decrypting it and obtaining a cleartext version of combined block **530**. Combined block **530** comprises basic authorization request **510**, a copy of merchant computer system's **130** encryption public key certificate **515** and a copy of merchant computer system's **130** signature public key certificate **520**, as well as merchant digital signature **525**.

In function block **620**, payment gateway computer system **140** verifies merchant computer system's **130** encryption public key certificate **515** and merchant computer system's **130** signature public key certificate **520**. Payment gateway computer system **140** performs this verification by making a call to the certification authorities associated with each certificate. If

verification of either certificate fails, payment gateway computer system **140** rejects the authorization request.

In function block **625**, payment gateway computer system **140** validates merchant digital signature **525**. Payment gateway computer system **140** performs this validation by calculating a message digest over the contents of the combined basic authorization request **510**, the encryption public key certificate **515** and the signature public key certificate **520**. Payment gateway computer system **140** then decrypts digital signature **525** to obtain a copy of the equivalent message digest calculated by merchant computer system **130** in function block **420**. If the two message digests are equal, the digital signature **525** is validated. If validation fails, payment gateway computer system **140** rejects the authorization request.

In function block **630**, payment gateway computer system **140** determines the financial institution for which authorization is required by inspection of basic authorization request **510**. Payment gateway computer system **140** contacts the appropriate financial institution using a secure means, e.g, a direct-dial modem-to-modem connection, or a proprietary internal network that is not accessible to third parties, and using prior art means, obtains a response indicating whether the requested payment is authorized.

Payment Authorization Response Generation

Function blocks **635** through **685** depict the steps of generating and transmitting a payment authorization request response. Figures **7A** through **7J** depict views of the payment authorization response and its component parts.

In function block **635**, payment gateway computer system **140** creates a basic authorization response **710**. The basic authorization request is a data area that includes all the information to determine whether a request was granted or denied. Figure **7A** depicts basic authorization response **710**.

In function block **640**, payment gateway computer system **140** combines basic authorization response **710**, and a copy of its signature public key certificate **720**. Payment computer system **140** calculates a digital signature **725** for the combined contents of the combined block **730** comprising basic authorization response **710** and the signature public key certificate **720**, and appends the signature to the combination of the combined basic authorization response **710** and the signature public key certificate **720**. The payment gateway computer system calculates digital signature **725** by first calculating a message digest based on the contents of the combined basic authorization response **710** and signature public key certificate **720**. The message digest is then encrypted using the merchant computer system's **140** digital signature private key, thus forming a digital signature.

Figure **7B** depicts the combined block **730** formed in function block **640** and containing basic authorization response **710**, the signature public key certificate **720**, and digital signature **725**.

In function block **645**, payment gateway computer system **150** generates a first symmetric random encryption key **740**, denoted as RK-1. Figure **7C** depicts first random encryption key RK-1 **740**.

In function block **650**, payment gateway computer system **140** encrypts combined block **730** using random encryption key RK-1 **740** to form encrypted combined block **750**. Figure **7D** depicts encrypted combined

block **750**. The encryption state of encrypted combined block **750** is graphically shown by random key lock **755**, which indicates that encrypted combined block **750** is encrypted using random key RK-1 **740**.

In function block **655**, payment gateway computer system **140** encrypts random encryption key RK-1 **740** using the public key of merchant computer system **130** to form encrypted random key RK **760**. Figure **7E** depicts encrypted random key RK-1 **760**. The encryption state of encrypted random key **760** is graphically shown by merchant public key lock **765**, which indicates that encrypted random key **760** is encrypted using the merchant public key.

In function block **660**, payment gateway computer system **140** generates a random capture token **770**. Random capture token **770** is utilized in subsequent payment capture processing to associate the payment capture request with the payment authorization request being processed. Figure **7F** depicts capture token **775**.

In function block **665**, payment gateway computer system **140** generates a second symmetric random encryption key **775**, denoted as RK-2. Figure **7G** depicts second random encryption key RK-2 **775**.

In function block **670**, payment gateway computer system **140** encrypts capture token **770** using random encryption key RK-2 **770** to form encrypted capture token **780**. Figure **7H** depicts encrypted capture token **780**. The encryption state of encrypted capture token **780** is graphically shown by random key lock **785**, which indicates that encrypted capture token **780** is encrypted using random key RK-2 **770**.

In function block **675**, payment gateway computer system **140** encrypts second random encryption key RK-2 **775** using its own public key to form encrypted random key RK-2 **790**. Figure **7I** depicts encrypted random key RK-2 **790**. The encryption state of encrypted random key **790** is graphically shown by payment gateway public key lock **795**, which indicates that encrypted random key **790** is encrypted using the payment gateway public key.

In function block **680**, payment gateway computer system **140** concatenates encrypted combined block **750**, encrypted random key RK-1 **760**, encrypted capture token **780** and encrypted random key RK-2 **790** to form merchant authorization response **325**. Figure **7J** depicts merchant authorization response **325** comprising encrypted combined block **750**, encrypted random key RK-1 **760**, encrypted capture token **780** and encrypted random key RK-2 **790**. In function block **685**, payment gateway computer system **140** transmits merchant authorization response **325** to merchant system **130**.

Payment Authorization Response Processing

Figure **8** depicts the detailed steps of processing a payment authorization response. In function block **810**, merchant computer system **130** applies its private key to encrypted random key RK-1 **760** contained within received merchant authorization response **325**, thereby decrypting it and obtaining a cleartext version of random key RK-1 **740**. In function block **820**, merchant computer system **130** applies random key RK-1 **740** to encrypted combined block **750**, thereby decrypting it and obtaining a cleartext version of combined block **730**. Combined block **730** comprises basic authorization response **710**, a copy of payment gateway computer system's **140** signature public key certificate **720**, as well as payment gateway digital signature **725**. In function block **830**, merchant computer system **130** verifies payment

gateway computer system's **140** signature public key certificate **720**. Merchant computer system **130** performs this verification by making a call to the certification authority associated with the certificate. If verification of the certificate fails, merchant computer system **130** concludes that the authorization response is counterfeit and treats it though the authorization request had been rejected.

In function block **840**, merchant computer system **130** validates payment gateway digital signature **725**. Merchant computer system **130** performs this validation by calculating a message digest over the contents of the combined basic authorization request **710** and the signature public key certificate **720**. Merchant computer system **130** then decrypts digital signature **725** to obtain a copy of the equivalent message digest calculated by payment gateway computer system **140** in function block **640**. If the two message digests are equal, the digital signature **725** is validated. If validation fails, concludes that the authorization response is counterfeit and treats it though the authorization request had been rejected.

In function block **850**, merchant computer system **130** stores encrypted capture token **780** and encrypted random key RK-2 **790** for later use in payment capture. In function block **860**, merchant computer system **130** processes the customer purchase request in accordance with the authorization response **710**. If the authorization response indicates that payment in authorized, merchant computer system **130** fills the requested order. If the authorization response indicates that payment is not authorized, or if merchant computer system **130** determined in function block **830** or **840** that the authorization response is counterfeit , merchant computer system **130** indicates to the customer that the order cannot be filled.

Payment Capture

Figure **9** depicts an overview of the method of securely supplying payment capture information to payment gateway **140** in order to obtain payment capture. In function block **910**, merchant computer system **130** generates a merchant payment capture request **915** and transmits it to payment gateway computer system **140**. In function block **930**, payment gateway system **140** processes the payment capture request **915**, generates a payment capture response **925** and transmits it to merchant computer system **130**. In function block **920**, merchant computer system **130** processes payment capture response **925** and verifies that payment for the goods or services sought to be obtained by the customer have been captured.

Payment Capture Request Generation

Figure **10** depicts the detailed steps of generating and transmitting a payment capture request. Figures **11A** through **11F** depict views of the payment capture request and its component parts. In function block **1010**, merchant computer system **130** creates a basic capture request **510**. The basic capture request is a data area that includes all the information needed by payment gateway computer system **140** to trigger a transfer of funds to the merchant operating merchant computer system **130**.

Specifically, a capture request includes a capture request amount, a capture token, a date, summary information of the purchased items and a Merchant ID (MID) for the particular merchant. Figure **11A** depicts basic authorization request **1110**.

In function block **1020**, merchant computer system **130** combines basic capture request **1110**, a copy of its encryption public key certificate **1115** and a copy of its signature public key certificate **1120**. Merchant computer system **130** calculates a digital signature **1125** for the combined contents of the combined block **1130** comprising basic capture request **1110**, the encryption public key certificate **1115** and the signature public key certificate **1120**, and appends it to the combination of the combined basic capture request **1110**, the encryption public key certificate **1115** and the signature public key certificate **1120**. The merchant computer system calculates digital signature **1125** by first calculating a message digest over the contents of the combined basic capture request **1110**, the encryption public key certificate **1115** and the signature public key certificate **1120**. The message digest is then encrypted using the merchant computer system's **130** digital signature private key, thus forming a digital signature.

Figure **11B** depicts the combined block **1130** formed by function block **1020** and containing basic capture request **1110**, the encryption public key certificate **1115**, the signature public key certificate **1120**, and digital signature **1125**. In function block **1030**, merchant computer system **130** generates a random encryption key **1140**, denoted as RK-3. Random encryption key RK-3 **1140** is a symmetric encryption key. Figure **11C** depicts random encryption key RK-3 **1140**. In function block **1040**, merchant computer system **130** encrypts combined block **1130** using random encryption key RK-3 **1140** to form encrypted combined block **1150**. Figure **11D** depicts encrypted combined block **1150**. The encryption state of encrypted combined block **1150** is graphically shown by random key lock **1155**, which indicates that encrypted combined block **1150** is encrypted using random key RK-3 **1140**. In function block **1050**, merchant computer system **130** encrypts random encryption key RK-3 **1140** using the public

key of payment gateway system **140** to form encrypted random key **1160**. Figure **11E** depicts encrypted random key **1160**. The encryption state of encrypted random key **1160** is graphically shown by payment gateway public key lock **1165**, which indicates that encrypted random key RK-3 **1160** is encrypted using the payment gateway public key.

In function block **1060**, merchant computer system **130** concatenates encrypted combined block **1150**, encrypted random key **1160**, and the encrypted capture token **780** and encrypted random key RK-2 **790** that were stored in function block **850** to form merchant capture request **915**. Figure **11F** depicts merchant capture request **915**, comprising encrypted combined block **1150**, encrypted random key **1160**, encrypted capture token **780** and encrypted random key RK-2 **790**. In function block **1070**, merchant computer system **130** transmits merchant capture request **915** to payment gateway system **140**.

Payment Capture Request Processing

Figure **12** depicts the detailed steps of processing a payment capture request and generating and transmitting a payment capture request response. Function blocks **1210** through **1245** depict the steps of processing a payment capture request, while function blocks **1250** through **1285** depict the steps of generating and transmitting a payment capture request response. In function block **1210**, payment gateway computer system **140** applies its private key to encrypted random key **1160** contained within received merchant capture request **915**, thereby decrypting it and obtaining a cleartext version of random key RK-3 **1140**. In function block **1215**, payment gateway computer system **140** applies random key RK-3 **1140** to encrypted combined block **1150**, thereby decrypting it and obtaining a cleartext version of combined block **1130**. Combined block

1130 comprises basic capture request **1110**, a copy of merchant computer system's **130** encryption public key certificate **1115** and a copy of merchant computer system's **130** signature public key certificate **1120**, as well as merchant digital signature **1125**. In function block **1220**, payment gateway computer system **140** verifies merchant computer system's **130** encryption public key certificate **1115** and merchant computer system's **130** signature public key certificate **1120**. Payment gateway computer system **140** performs this verification by making a call to the certification authorities associated with each certificate. If verification of either certificate fails, payment gateway computer system **140** rejects the capture request.

In function block **1225**, payment gateway computer system **140** validates merchant digital signature **1125**. Payment gateway computer system **140** performs this validation by calculating a message digest over the contents of the combined basic capture request **1110**, the encryption public key certificate **1115** and the signature public key certificate **1120**. Payment gateway computer system **140** then decrypts digital signature **1125** to obtain a copy of the equivalent message digest calculated by merchant computer system **130** in function block **1020**. If the two message digests are equal, the digital signature **1125** is validated. If validation fails, payment gateway computer system **140** rejects the capture request. In function block **1230**, payment gateway computer system **140** applies its private key to encrypted random key RK-2 **790** contained within received merchant capture request **915**, thereby decrypting it and obtaining a cleartext version of random key RK-2 **775**. In function block **1235**, payment gateway computer system **140** applies random key RK-2 **775** to encrypted capture token **780**, thereby decrypting it and obtaining a cleartext version of capture token **770**.

In function block **1240**, payment gateway computer system **140** verifies that a proper transaction is being transmitted between capture token **780** and capture request **1110**. A capture token contains data that the gateway generates at the time of authorization. When the authorization is approved, the encrypted capture token is given to the merchant for storage. At the time of capture, the merchant returns the capture token to the gateway along with other information required for capture. Upon receipt of the capture token, the gateway compares a message made of the capture request data and the capture token data and transmits this information over a traditional credit/debit network. If an improperly formatted transaction is detected, payment gateway computer system **140** rejects the capture request. In function block **1245**, payment gateway computer system **140** determines the financial institution for which capture is requested by inspection of basic capture request **1110**. Payment gateway computer system **140** contacts the appropriate financial institution using a secure means, e.g, a direct-dial modem-to-modem connection, or a proprietary internal network that is not accessible to third parties, and using prior art means, instructs a computer at the financial institution to perform the requested funds transfer.

Payment Capture Response Generation

Function blocks **1250** through **1285** depict the steps of generating and transmitting a payment capture request response. Figures **13A** through **13F** depict views of the payment capture response and its component parts.

In function block **1250**, payment gateway computer system **140** creates a basic capture response **710**. The basic capture request is a data area that includes all the information to indicate whether a capture request was granted or denied. Figure **13A** depicts basic authorization request **1310**.

In function block **1255**, payment gateway computer system **140** combines basic capture response **1310**, and a copy of its signature public key certificate **1320**. Payment computer system **140** calculates a digital signature **1325** for the combined contents of the combined block **1330** comprising basic capture response **1310** and the signature public key certificate **1320**, and appends the signature to the combination of the combined basic authorization request **1310** and the signature public key certificate **1320**. The payment gateway computer system calculates digital signature **1325** by first calculating a message digest over the contents of the combined basic capture response **1310** and signature public key certificate **720**. The message digest is then encrypted using the merchant computer system's **140** digital signature private key, thus forming a digital signature.

Figure **13B** depicts the combined block **1330** formed by function block **1255** and containing basic capture request **1310**, the signature public key certificate **1320**, and digital signature **1325**. In function block **1260**, payment gateway computer system **140** generates a symmetric random encryption key **1340**, denoted as RK-4. Figure **13C** depicts random encryption key RK-4 **1340**. In function block **1275**, payment gateway computer system **140** encrypts combined block **1330** using random encryption key RK-4 **1340** to form encrypted combined block **1350**. Figure **13D** depicts encrypted combined block **1350**. The encryption state of encrypted combined block **1350** is graphically shown by random key lock **1355**, which indicates that encrypted combined block **1350** is encrypted using random key RK-4 **1340**. In function block **1275**, payment gateway computer system **140** encrypts random encryption key RK-4 **1340** using the public key of merchant computer system **130** to form encrypted random key RK-4 **1360**. Figure **13E** depicts encrypted random key RK-4 **1360**. The

encryption state of encrypted random key **1360** is graphically shown by merchant public key lock **1365**, which indicates that encrypted random key **1360** is encrypted using the merchant public key. In function block **1280**, payment gateway computer system **140** concatenates encrypted combined block **1350** and encrypted random key RK-4 **1360** to form merchant capture response **925**. Figure **13F** depicts merchant capture response **925** comprising encrypted combined block **1350** and encrypted random key RK-4 **1360**. In function block **1285**, payment gateway computer system **140** transmits merchant capture response **925** to merchant system **130**.

Payment Capture Response Processing

Figure **14** depicts the detailed steps of processing a payment capture response. In function block **1410**, merchant computer system **130** applies its private key to encrypted random key RK-4 **1360** contained within received merchant capture response **925**, thereby decrypting it and obtaining a cleartext version of random key RK-4 **1340**. In function block **1420**, merchant computer system **130** applies random key RK-4 **1340** to encrypted combined block **1350**, thereby decrypting it and obtaining a cleartext version of combined block **1330**. Combined block **1330** comprises basic capture response **1310**, a copy of payment gateway computer system's **140** signature public key certificate **1320**, as well as payment gateway digital signature **1325**. In function block **1430**, merchant computer system **130** verifies payment gateway computer system's **140** signature public key certificate **1320**. Merchant computer system **130** performs this verification by making a call to the certification authority associated with the certificate. If verification of the certificate fails, merchant computer system **130** concludes that the capture response is counterfeit and raises an error condition.

In function block **1440**, merchant computer system **130** validates payment gateway digital signature **1325**. Merchant computer system **130** performs this validation by calculating a message digest over the contents of the combined basic authorization request **1310** and the signature public key certificate **1320**. Merchant computer system **130** then decrypts digital signature **1325** to obtain a copy of the equivalent message digest calculated by payment gateway computer system **140** in function block **1255**. If the two message digests are equal, the digital signature **1325** is validated. If validation fails, merchant computer system **130** concludes that the authorization response is counterfeit and raises an error condition. In function block **1450**, merchant computer system **130** stores capture response for later use in by legacy system accounting programs, e.g. to perform reconciliation between the merchant operating merchant computer system **130** and the financial institution from whom payment was requested, thereby completing the transaction. The system of the present invention permits immediate deployment of a secure payment technology architecture such as the SET architecture without first establishing a public-key encryption infrastructure for use by consumers. It thereby permits immediate use of SET-compliant transaction processing without the need for consumers to migrate to SET-compliant application software.

VIRTUAL POINT OF SALE (VPOS) DETAILS

A Virtual Point of Sale (VPoS) Terminal Cartridge is described in accordance with a preferred embodiment. The VPoS Terminal Cartridge provides payment functionality similar to what a Verifone PoS terminal ("gray box") provides for a merchant today, allowing a merchant to process payments securely using the Internet. It provides full payment functionality for a variety of payment instruments.

Payment Functionality

Figure **15A** illustrates a payment processing flow in accordance with a preferred embodiment. The payment functionality provided by the VPoS terminal is divided into two main categories: "Merchant-Initiated" **1510** and "Consumer-Initiated" **1500.** Some payment transactions require communication with the Acquirer Bank through the Gateway **1530.** The normal flow of a transaction is via the VPoS Cartridge API **1512** to the VPoS C++ API **1514** into the payment protocol layer **1516** which is responsible for converting into legacy format for utilization with existing host payment authorization systems. The output from the payment protocol layer **1516** is transmitted to the authorization processing center via the gateway **1530.** These transactions are referred to as "Online Transactions" or "Host Payments." The transactions that can be done locally by the merchant without having to communicate with the Acquirer Bank are referred to as "Local Functions and Transactions." To support different types of payment instruments, the VPoS Terminal payment functionality is categorized as set forth below.

- **Host Payment Functionality:** These transactions require communication with the final host, either immediately or at a later stage. For example, an Online Authorization-Only transaction, when initiated, communicates with the host immediately. However, an Off-line Authorization-Only transaction is locally authorized by the VPoS terminal without having to communicate with the host, but at a later stage this off-line authorization transaction is sent to the host. Within the Host Payment Functionality some transactions have an associated Payment Instrument, while others do not. These two kinds of transactions are:
- **Host Financial Payment Functionality:** These transactions have a Payment Instrument (Credit Card, Debit Card, E-Cash, E-Check, etc.)

associated with them. For example, the "Return" transaction, which is initiated upon returning a merchandise to the merchant.

- **Host Administrative Payment Functionality:** These transactions do not require a payment instrument, and provide either administrative or inquiry functionality. Examples of these transactions are "Reconcile" or the "Batch Close."
- **Local Functions and Transactions:** These transactions do not require communication with the host at any stage, and provide essential VPoS terminal administrative functionality. An example of this is the VPoS terminal configuration function, which is required to set up the VPOS terminal. Another example is the "VPoS Batch Review" function, which is required to review the different transactions in the VPoS Batch or the Transaction Log.

Payment Instruments

A preferred embodiment of a VPoS terminal supports various Payment Instruments. A consumer chooses a payment based on personal preferences. Some of the Payment Instruments supported include:

- Credit Cards
- Debit Cards
- Electronic Cash
- Electronic Checks
- Micro-Payments (electronic coin)
- Smart Cards

URL Table

The table below enumerates the URLs corresponding to the transactions supported by the VPoS Terminal Cartridge. Note that the GET method is allowed for all transactions; however, for transactions that either create or

modify information on the merchant server, a GET request returns an HTML page from which the transaction is performed via a POST method.

Transaction	URL	POST	Access Control
HOST FINANCIAL PAYMENT FUNCTIONALITY			
auth capture	/vpost/mi/authcapture/	allowed	merchant login/password
auth capture	/vpost/ci/authcapture/	allowed	no access control
auth only	/vpost/mi/authonly/	allowed	merchant login/password
auth only	/vpost/ci/authonly/	allowed	no access control
adjust	/vpost/mi/adjust/	allowed	merchant login/password
forced post	/vpost/mi/forcedpost/	allowed	merchant login/password
offline auth	/vpost/mi/offlineauth/	allowed	merchant login/password
offline auth	/vpost/ci/offlineauth/	allowed	no access control
pre auth	/vpost/mi/preauth/	allowed	merchant login/password
pre auth comp	/vpost/mi/preauthcomp/	allowed	merchant login/password
return	/vpost/mi/return	allowed	merchant login/password
return	/vpost/ci/return/	allowed	no access control

void	/vpost/mi/void/	allowed	merchant login/password

HOST ADMINISTRATIVE PAYMENT FUNCTIONALITY

balance inquiry	/vpost/mi/bi/	not allowed	merchant login/password
host logon	/vpost/mi/hostlogon/	allowed	merchant login/password
parameter download	/vpost/mi/parametersdnld/	not allowed	merchant login/password
reconcile	/vpost/mi/reconcile/	allowed	merchant login/password
test host	/vpost/mi/testhost/	not allowed	merchant login/password

LOCAL FUNCTIONS & TRANSACTIONS

accum review	/vpost/mi/accum/review/	not allowed	merchant login/password
batch review	/vpost/mi/batch/review/	not allowed	merchant login/password
cdt review	/vpost/mi/cdt/review/	not allowed	merchant login/password
cdt update	/vpost/mi/cdt/update/	allowed	merchant login/password
cpt review	/vpost/mi/cpt/review	not allowed	merchant login/password
cpt update	/vpost/mi/cpt/update/	allowed	merchant login/password
clear accum	/vpost/accum/clea	allowed	merchant

	r/		login/password
clear batch	/vpost/mi/batch/clear/	allowed	merchant login/password
hdt review	/vpost/mi/hdt/review/	not allowed	merchant login/password
hdt update	/vpost/mi/hdt/update/	allowed	merchant login/password
lock vpos	/vpost/mi/lock/	allowed	merchant login/password
query txn	/vpost/ci/querytxn/	not allowed	no access control
query txn	/vpost/mi/querytxn/	not allowed	merchant login/password
tct review	/vpost/mi/tct/review/	not allowed	merchant login/password
tct update	/vpost/mi/tct/update/	allowed	merchant login/password
unlock vpos	/vpost/mi/unlock/	allowed	merchant login/password

URL Descriptions

This section describes the GET and POST arguments that are associated with each transaction URL. It also describes the results from the GET and POST methods. For URLs that produce any kind of results, the following fields are present in the HTML document that is returned by the VPoS Terminal Cartridge:

txnDate	Date of the transaction (mm/dd/yy or dd/mm/yy)
txnTime	Time of the transaction (hh:mm:ss GMT or hh:mm:ss local time)
merchantId	Merchant ID of the merchant using the VPoS terminal
terminalId	VPoS Terminal Id
txnNum	Transaction number of the given transaction
txnType	Type of transaction

For URLs that deal with financial transactions, the following fields are present in the HTML document that is returned by the VPoS terminal cartridge:

txnAmount	Transaction amount that is being authorized, forced posted, voided, etc.
poNumber	Purchase order number
authIdentNum	Authorization ID number for the transaction
retRefNum	Retrieval reference number for the given transaction

piInfo	Payment instrument information. This varies for different payment instruments. For example, in the case of credit cards, the credit card number (piAcctNumber) and expiration date (piExpDate) are returned.

Accumulate Review

URL Functionality: This is a local information inquiry function that retrieves the local (merchant's) transaction totals (accumulators).

GET Arguments: None.

GET Results: Retrieves the transaction totals for the merchant. Currently, the total is returned as an HTML document. The transaction totals currently returned are:

creditAmt	Total Credit Amount since the last settlement logged in the VPoS terminal
creditCnt	Total Credit Count since the last settlement logged in the VPoS terminal
debitAmt	Total Debit Amount since the last settlement logged in the VPoS terminal
debitCnt	Total Debit Count since the last settlement logged in the VPoS terminal

Note: Accum Review is a local function, as opposed to Balance Inquiry which is done over the Internet with the host.

Adjust

URL Functionality: Corrects the amount of a previously completed transaction.

GET Arguments: None

GET Results: Because the Adjust transaction modifies data on the merchant server, the POST method should be used. Using the GET method returns an HTML form that uses the POST method to perform the transaction.

POST Arguments:

pvsTxnNum	Previous transaction number
txnAdjustedAmount	The adjusted transaction amount. Note that the original transaction amount is easily retrievable from the previous transaction number.

POST Results: On success, pvsTxnNum and txnAdjustedAmount are presented in the HTML document, in addition to the transaction fields described above.

Auth Capture

URL Functionality: This transaction is a combination of Auth Only (Authorization without capture) and Forced Post transactions.

GET Arguments: None

GET Results: Because the Auth Capture transaction modifies data on the merchant server side, the POST method should be used. Using the GET method returns an HTML form that uses the POST method to perform the transaction.

POST Arguments:

piAcctNumber	Payment Instrument account number, e.g., Visa credit card number

piExpDate	Expiration date
txnAmt	Transaction amount

POST Results: On success, an HTML document that contains the transaction fields described above is returned. On failure, an HTML document that contains the reason for the failure of the transaction is returned. The transaction is logged into a VPoS Terminal transaction log for both instances.

Auth Only

URL Functionality: Validates the cardholder's account number for a Sale that is performed at a later stage. The transaction does not confirm the sale to the host, and there is no host data capture. The VPoS captures this transaction record and later forwards it to confirm the sale in the Forced Post transaction request.

GET Arguments: None.

GET Results: Because the Auth Only transaction modifies data on the merchant server side, the POST method should be used. Using the GET method returns an HTML form that uses the POST method to perform the transaction.

POST Arguments:

piAcctNum ber	Payment Instrument account number, e.g., Visa credit card number
piExpDate	Expiration date
txnAmt	Transaction amount

POST Results: On success, an HTML document that contains the transaction fields is returned. On failure, an HTML document that contains

the reason for the failure of the transaction is returned. The transaction is logged into VPoS Terminal transaction log for both instances.

NOTE: The /vpost/ci/authonly/ URL should be used for customer-initiated transactions. /vpost/mi/authonly/ should be used for merchant-initiated transactions.

Balance Inquiry

URL Functionality: Performs an on-line inquiry or the merchant's balance.

GET Arguments: None

GET Results:

mrchtBlnce Amt	Merchant balance amount for a given merchant. The balance amount at any given time is the difference between the credit and debit amount since the last settlement between the merchant and the acquirer.

Batch Review

URL Functionality: Retrieves all records from the transaction log or the batch.

GET Arguments: None

GET Results: The GET method retrieves the transactions that have been batched in the VPoS terminal for future reconciliation. The batch can be cleared from the VPoS terminal after a manual reconciliation between the acquirer and the VPoS. The batch data is retrieved as a set of records and is formatted as a table in the HTML document. The following fields are present in a typical record:

nTransType	Transaction type
nPurchOrder No	Purchase order number
szAcctNum	Customer's payment instrument account number
szExpDate	Customer's payment instrument expiration date
szTransAmt	Transaction amount
szTransDate	Transaction date
szTransTime	Transaction time
szRetrievalRef Num	Transaction's retrieval reference number
szAuthId	Authorization ID for the transaction
szOrigAmt	Original transaction amount
szBatchNum	Batch number for the given transaction
nCurrencyTyp e	Currency in which the transaction was done
lnTransNum	Transaction number

CDT Review

URL Functionality: Displays the VPoS terminal configuration data corresponding to the Card Definition Table (CDT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be modified and posted using the /vpost/mi/cdt/update/ URL to update the card definition table. Not all fields in the card definition table are editable. The following fields are returned in a form to the user:

nHostIndex	Index into the Host Definition Table or the Acquirer that maps to this card issuer.
szPANLo	Low end of the PAN (Primary Account Number) range
szPANHi	High end of the PAN range
nMaxPANDigit	Maximum number of digits in the PAN for this acquirer.
NMinPANDigit	Minimum number of dits in the PAN for the acquirer
szCardLabel	Card Issuer's name
Transactions Available bit vector	Specifies if a particular transaction is allowed for a given card range.

(Some of these fields are not editable by a merchant, and still need to be determined.)

CDT Update

URL Functionality: Updates the VPoS terminal configuration data corresponding to the Card Definition Table (CDT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be filled out and posted using the /vpost/mi/cdt/update URL to update the card definition table.

POST Arguments: (Editable CDT fields need to be decided.)

POST Results: (Depends on editable CDT fields, and therefore needs to be decided.)

Clear Accumulator

URL Functionality: Zeroes out the accumulator totals currently resident in the VPoS terminal.

GET Arguments: None.

GET Results: Presents a form that uses the POST method to zero the accumulators.

POST Arguments: None.

POST Results: Zeroes the accumulators/transaction totals in the VPoS terminal.

Clear Batch

URL Functionality: Zeroes out the transaction logs currently batched in the VPoS terminal.

GET Arguments: None.

GET Results: Presents a form that uses the POST method to clear the batch.

POST Arguments: None.

POST Results: Zeroes the transactions that comprise the batch in the VPoS terminal.

Forced Post

URL Functionality: Confirms to the host the completion of a sale, and requests for data capture of the transaction. This is used as a follow-up transaction after doing an Authorization (Online or Off-line) transaction.

GET Arguments: None.

GET Results: Returns the HTML form for performing the Forced Post transaction.

POST Arguments:

pvsTxnNum	the previous transaction number from an auth only transaction

POST Results: On success, pvsTxnNum is presented in the HTML document. On failure, an HTML document is returned that contains the reason for the failure of the transaction.

HDT Review

URL Functionality: Displays the VPoS terminal configuration data corresponding to the Host Definition Table (HDT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be modified and posted using the /vpost/mi/hdt/update URL to update the hosts definition table. Not all fields in the host definition table are editable. The following fields are returned in a form to the user:

szTermId	Terminal ID for this VPoS terminal
szMerchId	Merchant ID for this VPoS terminal
szCurrBatchNum	Current batch number existing on the VpoS
szTransNum	Reference number for the next transaction in the VPoS transaction log/batch. This is generated by VPoS and is not editable by the

	merchant.
szTPDU	Transport Protocol Data Unit. Required for building the ISO 8583 packet.
InSTAN	System trace number; message number of the next transaction to be transmitted to this acquirer.
szNII	Network International Number. Required for building the ISO 8583 packet.
szHostName	Name for identifying the host.
nHostType	Host type
nNumAdv	Number of off-line transactions that can be piggy-backed at the end of an on-line transaction.
Data Capture Required Bit vector:	Specifies for which transactions data capture is required.

(Some of these fields are not editable by a merchant and need to be determined.)

HDT Update

URL Functionality: Updates the VPoS terminal configuration data corresponding to the Host Definition Table (HDT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be filled out and posted to the merchant server using the /vpost/mi/hdt/update URL to update the host definition table

Unlock VPOS

URL Functionality: Local function that starts the VPOS at the start of the day.

GET Arguments: None.

GET Results: Returns an HTML form that uses the POST method to perform this transaction.

POST Arguments: None.

POST Results: Resets a Boolean flag on the merchant server that enables transactions to be accepted by the VPoS terminal.

Offline Auth

URL Functionality: This transaction is same as the "Authorization Only" transaction, except that the transaction is locally captured by the VPoS terminal without having to communicate with the host. A Forced Post operation is done as a follow-up operation of this transaction.

GET Arguments: None.

GET Results: Because the Offline Auth transaction modifies data on the merchant server side, the POST method should be used. Using the GET method returns an HTML form for using the POST method to perform the transaction.

POST Arguments:

piAcctNum ber	Payment Instrument account number, e.g., Visa credit card number
piExpDate	Expiration date
txnAmt	Transaction amount

POST Results: On success, an HTML document that contains the transaction fields described in Section 4.1 is returned. On failure, an HTML document that contains the reason for the failure of the transaction is

returned. The transaction is logged into VPoS terminal transaction log for both instances.

Parameter Download

URL Functionality: Downloads the VPoS configuration information from the host and sets up the VPOS in the event of the configuration data being changed.

GET Arguments: None

GET Results: Retrieves an HTML form that uses the POST method for the parameter download transaction.

POST Arguments: None.

POST Results: Downloads the following parameters from the host and uploads them into the VPoS terminal configuration table.

- card/issuer definition table (CDT)
- host/acquirer definition table (HDT)
- communications parameter table (CPT)
- terminal configuration table (TCT)

The various configuration parameters can be reviewed and modified using the URLs for the desired functionality.

Pre Auth

URL Functionality: Used in lodging and hotel establishments to pre-authorize a charge that is completed some time in future.

GET Arguments: None

GET Results: Retrieves the HTML form for posting the pre-authorizaton transaction.

POST Arguments:

piAcctNum ber	Payment Instrument account number, e.g., Visa credit card number
piExpDate	Expiration date

Pre Auth Comp

URL Functionality: Completes a pre-authorization transaction.

GET Arguments: None

GET Results: Retrieves the HTML form for posting the pre-authorization completion transaction.

POST Arguments:

pvsTxnNum	Previous transaction number from an auth only transaction

POST Results: On success, pvsTxnNum is presented in the HTML document. On failure, an HTML document is returned that contains the reason for the failure of the transaction.

Reconcile

URL Functionality: This transaction is done at the end of the day to confirm to the host to start the settlement process for the transactions captured by the host for that particular VPoS batch.

GET Arguments: None

GET Results: Retrieves the HTML form for posting the Reconcile transaction.

POST Arguments: None.

POST Results: On success, the reconcile function prints any discrepancies in the merchant's batch of transactions and totals vis-a-vis the host's batch

of transactions in totals. The output format is a combination of the output of the Batch Review and Accum Review transactions.

Return

URL Functionality: Credits the return amount electronically to the consumer's account when previously purchased merchandise is returned. The VPoS terminal captures the transaction record for this transaction.

GET Arguments: None

GET Results: Retrieves the HTML form for posting the Return transaction.

POST Arguments:

prevTxnNum	Reference to the previous transaction number

The previous transaction has access to the following fields:

txnAmount	Transaction amount
piAccountNum	Payment instrument account number
piExpDate	Payment instrument expiration date

POST Results: On success, pvsTxnNum is presented in the HTML document, in addition to

Test Host

URL Functionality: Checks the presence of the host and also the integrity of the link from the VPoS to the host.

GET Arguments: None.

GET Results: On success, an HTML document is returned that reports success in connecting to the host. On failure, an HTML document is returned that reports the error encountered in testing the host.

Lock VPOS

URL Functionality: This local function locks or stops the VPoS terminal from accepting any transactions.

GET Arguments: None.

GET Results: Returns an HTML form that posts the locking of the VPoS terminal.

POST Arguments: None.

POST Results: On success, an HTML document is returned that contains the status that VPoS terminal was successfully. On failure, an HTML document is returned that reports the cause of failure of the operation, e.g., access denied, the VPoS terminal is already locked or is presently processing a transaction, etc.

Void

URL Functionality: Cancels a previously completed draft capture transaction.

GET Arguments: None.

GET Results: Retrieves an HTML form for posting the Void transaction.

POST Arguments:

pvsTxnNum	Transaction number from a previous Auth Only transaction.

Host Logon

URL Functionality: Administrative transaction used to sign-on the VPoS with the host at the start of the day, and also to download encryption keys for debit transactions.

GET Arguments: None

GET Results: Retrieves an HTML form for posting the Host Logon transaction.

POST Arguments: None.

POST Results: Currently, debit card based transactions are not supported. The result is an HTML document indicating the success or failure of the host logon operation.

CPT Review

URL Functionality: Returns the VPoS terminal configuration data corresponding to the Communications Parameter Table (CPT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values corresponding to the VPoS terminal's communication parameters. The form can be filled out and posted to the merchant server using the /vpost/mi/cpt/update URL to update the communications parameter table. The following fields are returned in a form to the user:

szAcqPriAddress	Primary Host address
szAcqSecAddress	Secondary Host address
szActTerAddress	Tertiary Host address

nRespTimeOut	Time-out value (in seconds) before which the VPoS should receive a response from the host

CPT Update

URL Functionality: Updates the VPoS terminal configuration data corresponding to the Communications Parameter Table (CPT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be modified and posted to update the communication parameter table.

POST Arguments:

szAcqPriAddress	Primary Host address
szAcqSecAddress	Secondary Host address
szActTerAddress	Tertiary Host address
nRespTimeOut	Time-out value (in seconds) before which the VPoS should receive a response from the host

POST Results: On success, the HTML document returned by the VPoS contains the values set by the merchant. On failure, the HTML document contains the reason for the failure of the invocation of the URL.

TCT Review

URL Functionality: Returns the VPoS terminal configuration data corresponding to the Terminal Configuration Table (TCT).

GET Arguments: None.

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be filled out and posted using the /vpost/mi/tct/update URL to update the terminal configuration table. The following fields are returned in a form to the user:

szMerchName	Merchant name
szSupervisorPwd	Supervisor password
fvPOSLock	1= VPoS locked, 0 = VPoS unlocked
szAuthOnlyPwd	Password for initiating auth-only transaction
szAuthCaptPwd	Password for initiating auth with capture transaction
szAdjustPwd	Password for adjust transaction
szRefundPwd	Password for refund transaction
szForcedPostPwd	Password for forced post transaction
szOfflineAuthPwd	Password for offline auth transaction
szVoidPwd	Password for void transaction
szPreAuthPwd	Password for pre-authorization transaction
szPreAuthCompPwd	Password for pre-authorization completion

TCT Update

URL Functionality: Updates the VPoS terminal configuration data corresponding to the Terminal Configuration Table (TCT).

GET Arguments: None

GET Results: The GET method returns a default HTML form that contains the current configuration values. The form can be filled out and posted using the /vpost/mi/tct/update URL to update the terminal configuration table.

POST Arguments: All arguments in TCT Review functionality are the returned values from the /vpost/mi/tct/update the URL.

szMerchName	Merchant name
szSupervisorPwd	Supervisor password
fvPOSLock	1= VPoS locked, 0 = VPoS unlocked
szAuthOnlyPwd	Password for initiating auth-only transaction
szAuthCaptPwd	Password for initiating auth with capture transaction
szAdjustPwd	Password for adjust transaction
szRefundPwd	Password for refund transaction
szForcedPostPwd	Password for forced post transaction
szOfflineAuthPwd	Password for offline auth transaction
szVoidPwd	Password for void transaction
szPreAuthPwd	Password for pre-authorization transaction
szPreAuthCompPwd	Password for pre-authorization completion

POST Results: On success, the POST modifies values of the terminal configuration table parameters. On failure, the HTML document contains the reason for the failure of the transaction.

Query Transactions

URL Functionality: Permits the merchant and customer to query a given transaction corresponding to a transaction number.

GET Arguments: -

txnNum	Transaction number

GET Results: For a given transaction, the URL returns an HTML document. If a transaction refers to an older transaction, the transaction's entire history is made available.

URL results

Depending upon the method (GET/POST) as well as the success or failure of the HTTP request, different documents are returned to the user. The VPoS terminal provides a framework whereby different documents are returned based upon a number of preferences. Currently the language and content-type are supported as preferences.

A simple framework is proposed here. Each of the transaction has a set of documents associated with it: form for the payment transaction, GET success, GET failure, POST success, and POST failure.

In the directory structure defined below, documents are stored corresponding to the preferences. The top level of the directory structure is the content-type, the next level is language (for NLS support). For example, to create text/html content in US English & French, the directory structure given below would contain the HTML documents for each of the transactions. The VPoS terminal cartridge has a configuration file that allows the user to specify the content-type as well as the language to be

used for a cartridge. The first release of the VPoS terminal cartridge supports one content-type and language for each server.

Data Structures & Functions

Functions

A brief description of the Virtual Point of Sale Terminal cartridge functions are provided below. VPosTInit(), VPosTExec() and VPosTShut() are the entry points required for each cartridge in accordance with a preferred embodiment. The other functions implement some of the key VPoST cartridge functionality.

VPosTInit()

```
/* VPosT cartridge Initialization here */
WRBReturnCode
VPosTInit( void **clientCtx ){
    vPosTCtx *vPosTCxp ;
    /* Allocate memory for the client context */
    if (!(vPosTCxp = (vPosTCtx *)malloc(sizeof(vPosTCtx))))
            return WRB_ERROR ;
      *clientCtx = (void *)vPosTCxp ;
      return (WRB_DONE) ;}
```

VPosTShut()

```
WRBReturnCode
VPosTShut( void *WRBCtx, void *clientCtx ){
      *WRBCtx ;  /* not used */
      assert( clientCtx ) ;
      /* Free the client context allocated in VPosTInit() routine
```

```
        free( clientCtx ) ;

    return (WRB_DONE) ;}
```

VPosTExec()

```
/* The driver cartridge routine */
WRBReturnCode
VPosTExec( void *WRBCtx, void *clientCtx )
{
        vPoSTCtx    *vPosTCxp ;
        char  *uri ;
        char  *txnMethod ;        /* HTTP method */
        enum eVPoSTTxn *txn ; /* VPosT transaction */
        char  *txnOutFile ; /* Output file from transaction */
        char  **txnEnv ; /* environment variables values for transaction */
        char  *txnContent ; /* transaction's POST data content */
        WRBEntry  *WRBEntries ;
        int             numEntries;
        vPosTCxp = (vPosTCtx *) clientCtx ;
        /* WRBGetURL gets the URL for the current request */
        if (!(uri = WRBGetURL( WRBCtx )))
                return (WRB_ERROR) ;
        /* WRBGetContent() gets the QueryString/POST data content */
        if (!(txnContent = WRBGetContent( WRBCtx ))) {
                return WRB_ERROR ;
        }
        /* WRBGetParserContent() gets the parsed content */
        if (WRB_ERROR == WRBGEtParsedContent( WRBCtx, &WRBEntries,
         &numEntries)) {
                return WRB_ERROR ;
```

```
    }

    /* WRBGetEnvironment() gets the HTTP Server Environment */
    if (!(txnEnv = WRBGetEnvironment( WRBCtx ))) {
        return WRB_ERROR ;
    }
        /* VPosTGetMethod() gets the method for the current request */
        if (!(method = VPosTGetMethod( txnEnv ))){
                return (WRB_ERROR) ;
        }

        /* VPosTGetTxn() gets the VPosT transaction for the request */
        txn = VPosTGetTxn( uri );
        if (eTxnError == txn) {
                return (WRB_ERROR) ;
        }

        /* VPosTExecuteTransaction() executes the VPosT transaction */
        txnOutFile = VPosTExecuteTransaction( WRBCtx, txn, txnMethod,
          txnEnv, txnContent ) ;
        if (!(txnOutFile)) {
                return (WRB_ERROR) ;
        }
        /* Write out the file */
        VPosTWriteFile( txnOutFile ) ;
        return (WRB_DONE) ;
}
```

VPosTGetTxn()

```
enum eVPosTTxn
VPosTGetTxn( char *uri )
{
    /*
    * The function scans the uri and extracts the string
    * corresponding to the transaction and returns it to the
    * caller.
    */
}
```

Transaction Log format

This section describes the format of a record for the transaction log for the VPosT cartridge.

Field Name	**Field Description**
nTransType	Transaction Type
nPurchOrderNo	Purchase Order Number
szAcctNum	Payment Instrument Account number
szExpDate	Payment instrument expiration date
szTransAmt	Transaction amount
szTransDate	Date of transaction (configurable to be mm/dd/yy or dd/mm/yy)
szTransTime	Time of transaction (configurable to be GMT or local time)
szRetrievalRefNum	Retrieval reference number
szAuthId	Authorization ID

szOrigAmt	Original transaction amount
szBatchNum	Batch number to which this particular transaction belongs in the VPoST batch
nCurrencyType	Currency
lnTransNum	Transaction number

In the block diagram shown in Figure **15B**, the VPOS provides an interface for transactions which are initiated both by the consumer and the merchant. The merchant initiates a transaction from a Graphical User Interface (GUI) **1550** and all the transactions that are initiated by the consumer are routed by the Merchant WEB Server **1545**.

The Authorization/Data Capture Module **1560** processes the requests originated by the merchant or the consumer and routes them to the Protocol Module **1565**. The Protocol Module is responsible for building the payment protocol request packet (e.g., an SSL-encapsulated ISO 8583 packet) **1570** before sending the request to the Gateway **1579**. Then, the Gateway **1579** awaits a response from the Protocol Module **1565**, and upon receiving the response, the Gateway **1579** parses the data and provides unwrapped data to the Authorization/Data-Capture Module **1560**. The Authorization/Data-Capture Module **1560** analyzes the response and updates the Transaction Log **1580**. The Transaction Log **1580** contains information concerning any successfully completed transactions and the accumulators or the transaction totals. The VPOS terminal creates and maintains the Transaction Log **1580**, and the VPOS Configuration Data **1585** contains information which is used to configure the behavior of the VPOS.

The entire VPOS functionality is thread-safe and hence using the VPOS in a multi-threaded environment does not require any additional interfacing requirements.

Payment Functionality

As discussed above, the different Payment Functionality provided by the VPOS terminal can be divided into two main categories as "Merchant Initiated" and "Consumer Initiated." Some of these transactions require communication with the Gateway and these transactions are referred to as "Online Transactions." The transactions which can be done locally to the merchant without having to communicate are referred to as "Local Functions/Transactions." In order to provide support for many different types of Payment Instruments, the VPOS Payment Functionality have been categorized.

Host payment functionality and transactions require communication with the host either immediately or at a later stage. Each of the host financial payment transactions come to this category and require a Payment Instrument. These transactions can be initiated with different types of Payment Instruments which the VPOS terminal supports.

An authorization without capture transaction is used to validate the card holder's account number for a sale that needs to be performed at a later stage. The transaction does not confirm a sale's completion to the host, and there is no host data capture in this event. The VPOS captures this transaction record and later forwards it to the host to confirm the sale in a forced post transaction request. An authorization without capture transaction can be initiated both by the consumer and the merchant.

A forced post transaction confirms to a host computer that a completion of a sale has been accomplished and requests data capture of the transaction. The forced post transaction is used as a follow-up transaction after doing an authorization (Online or Off-line) transaction. The transaction can be initiated only by the merchant.

The authorization with post transaction is a combination of authorization without capture and forced post transactions. This transaction can be initiated both by the consumer and the merchant.

The offline post transaction is identical to the "authorization without capture" transaction, except that the transaction is locally captured by the VPOS without initiating communication with a host. A forced post operation is done as a follow-up operation of this transaction. This transaction can be initiated by both the consumer and the merchant.

The return transaction is used to credit the return amount electronically to the consumer's account when a purchased merchandise is returned. The VPOS captures the return transaction record when the merchandise is returned, and this transaction can be initiated only by the merchant.

The void transaction cancels a previously completed draft capture transaction. The VPOS GUI provides an interface for retrieving a transaction record required to be voided from the batch and passes it to the Authorization/Data-Capture module after confirmation. The batch record is updated to reflect the voided transaction after getting an approval from the gateway. This transaction can be initiated only by the merchant.

The pre-authorization transaction is identical to the authorization without capture transaction, but the consumers' "open-to-buy" amount is reduced by the pre-authorization amount. An example of this type of transaction is the "check-in" transaction in a hotel environment. A check-in transaction sends a pre-authorization request to the host, so that an amount required for the customers' stay in the hotel is reserved. The pre-authorization transaction is followed by a pre-authorization complete transaction. This transaction can be initiated both by the consumer and the merchant.

The pre-authorization complete transaction is done as a follow-up to the pre-authorization transaction. This transaction informs the host of the actual transaction amount. The pre-authorization complete transaction amount could be more or less than the pre-authorization amount. An example is the "check-out" transaction in a hotel environment. The check-out amount can be less than or more than the check-in amount. This transaction can only be initiated by a merchant.

The adjust transaction is initiated to make a correction to the amount of a previously completed transaction. The adjust transaction can be initiated only by the merchant. The host administrative transactions do not require any payment instrument. The balance inquiry transaction is used for on-line inquiry into the balance of the merchant's account. The batch data or the configuration data is not affected by this transaction.

The reconciliation or close transaction is processed at the end of the day to start the settlement process for the transactions captured by the host for that particular VPOS.

The host log-on transaction is an administrative transaction which is used to synchronize the VPOS with the host at the start of the day and also initiate a fresh batch at the VPOS terminal.

The parameters download transaction is used to download the VPOS configuration information from the host and set-up the VPOS in the event of any change in the configuration data. A test transaction is used to detect the presence of a host and the status of a link from the VPOS to the host.

Local transactions or functions are initiated by a merchant and do not require communication with the gateway. These transactions can only be initiated by a merchant. The totals or accumulators review is a local information inquiry function and is used to retrieve the local (merchant's) totals. The detail transaction or the batch review function is used to retrieve all the records from the transaction log or the batch. The clear batch function is used to start a fresh batch. This transaction is utilized to electronically reconcile the VPOS with the host and to manually reconcile the VPOS with the host. After completing the manual reconciliation processing, the merchant can initiate this transaction to start a fresh batch.

The clear accumulator function is similar to the clear batch functionality and resets all VPOS terminal accumulators to zero. This function is required when the merchant is not able to reconcile the VPOS with the host electronically.

The VPOS unlock or start transaction is a local function used to start the VPOS at the start of the day. The VPOS lock or stop function is used to Lock or stop the VPOS from accepting any transactions. The VPOS configuration setup function is used to setup the VPOS configuration data.

The VPOS configuration data is divided into different tables, for example, the Card/Issuer Definition Table (CDT), the Host/Acquirer Definition Table (HDT), the Communications Parameters Table (CPT) and the Terminal Configuration Table (TCT). The following sections explain each of these configuration tables in detail.

Host Definition Table (HDT)

The table contains information specific to the acquirer.

Field	**Attributes / Bytes**	**Field Description/Comments**
Terminal Identifier	ANS(20)	Terminal ID for this acquirer/host
Merchant Identifier	ANS(20)	Merchant ID for this acquirer/host
Current Batch Number	N(6)	Batch Number for the batch currently existing on the VPOS
Transaction Number	I(2)	Reference Number for next transaction in the VPOS transaction log/batch (VPOS generated)
TPDU	AN(10)	Transport Protocol Data Unit - Required for building the ISO 8583 packet.
STAN	L(4)	Systems Trace Number - Message Number of the transaction to be transmitted next for this acquirer.
NII	N(3)	Network International Identifier - Required for building the ISO 8583 packet.

Host Name or Label	ANS(20)	Name for identifying the host, e.g., "AMEX-SIN". This is only a text string and is used for the purpose of identifying the host.
No. of advice messages	I(2)	No. of off-line transactions (advice messages) that can be piggy-backed at the end of an on-line transaction. If set to zero then piggy-backing is disabled.

The following fields specify whether Data Capture Required for a particular transaction for this acquirer.

Field	**Attributes / Bytes**	**Field Description/Comments**
Host Protocol Type	I(2)	Host Protocol type, e.g., ISO 8583, SET, etc.,
Host Protocol Sub-Type	I(2)	Sub protocol type, e.g., AMEX-ISO8583, MOSET, etc.,
Auth Only DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Auth Capture DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Adjust DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Refund DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Cash Advance DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Cash Back DC	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED

Flag		
Off-line Auth DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Void DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Pre-Auth DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED
Pre-Auth Complete DC Flag	Bit(1 bit)	1 = REQUIRED, 0 = NOT REQUIRED

Card Definition Table (CDT)

This table contains information which are specific to the card issuer.

Field	**Attributes / Bytes**	**Field Description/Comments**
Host Index	I(2)	Index into the HDT or the acquirer which maps to this card issuer.
PAN Low Range	N(19)	Low end of the PAN range .
PAN High Range	N(19)	High end of the PAN range.
Minimum PAN digits	I(2)	The minimum number of digits in the PAN for this acquirer.
Maximum PAN digits	I(2)	The maximum number of digits in the PAN for this acquirer.
Card Label	ANS(20)	Card Issuer Name for identification, e.g., VISA.

5

The following fields specify whether a particular transaction is allowed for a card range.

Field	**Attributes / Bytes**	**Field Description/Comments**
Auth Only Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Auth Capture Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Adjust Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Refund Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Cash Advance Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Cash Back Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Off-line Auth Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Void Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Pre-Auth Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED
Pre-Auth Complete Allowed	Bit(1 bit)	1 = ALLOWED, 0 = NOT ALLOWED

Communications Parameter Table (CPT)

This table contains communications parameters information specific to an acquirer. The HDT and this table have a one-to-one mapping between them.

Field	Attributes / Bytes	Field Description/Comments
Primary Address	AN(100)	Primary Host Address (Telephone number, IP address, etc.)
Secondary Address	AN(100)	Secondary Host Address to be used if the Primary Address is busy or not available.
Tertiary Address	AN(100)	Tertiary Host Address.
Response Time-out	I(2)	Time-out value (in seconds) before which the VPOS should receive a response from the host.

Terminal Configuration Table (TCT)

This table contains information specific to a particular VPOS terminal.

Field	Attributes / Bytes	Field Description/Comments
Merchant Name	ANS(100)	Name of the merchant having the VPOS terminal.
VPOS Lock Flag	Bit (1 bit)	1 = VPOS Locked, 0 = VPOS Unlocked

Payment Instruments

As discussed above, the VPOS terminal supports different Payment Instruments and each of the Payment Functions described above can be initiated by these different Payment Instruments. The consumer making a purchase from a merchant provides a choice of payment methods depending

upon their personal preference. The Payment Instrument Class Hierarchy which is used by the different VPOS terminal Payment Functions is described below.

Message Sequence Diagram

Figure **17** shows a typical message flow between the consumer, merchant, VPOS terminal and the Gateway. This section describes the different classes listed in the previous section, their data and members, and defines the type of the transaction that is to be performed. Processing commences at **1700** when a merchant server receives a sales order and passes it via the VPoS Graphical User Interfece (GUI) **1710** to an authorizer **1720** for approval and subsequent protocol processing **1730** and ultimately transmission via the gateway **1740** to the network.

Class Name :

CVPCLTransaction

Data :

Transaction Type (int)

Transaction Date and Time (CPCLDateTime)

Card Definition Table (CVPCL_CDT)

Host Definition Table (CVPCL_HDT)

Communications Parameters Table (CVPCL_CPT)

Terminal Configuration Parameters (CVPCL_TCT)

Batch Record (CVPCLBatch)

Accumulator Record (CVPCLAccum)

Member Functions :

CVPCLTransaction();

EStatus GetTransType();

EStatus GetTransDateTime(CPCLDateTime&);

```
EStatus SetTransType(const int);
virtual EStatus InitializeTrans(TVPosParamsBlk *) = 0;
virtual EStatus ExecuteTrans(TVPosResultsBlk *) = 0;
virtual EStatus ShutDown() = 0;
```

Host Transaction Class Definitions

This section contains all the host transaction class definitions.

Host Transaction Class (CVPCLHostTrans)

This is an abstract base class derived from the CVPCLTransaction class and is used for deriving transaction classes which need to communicate with the host either immediately or at a later stage.

Class Name :

CVPCLHostTrans

Data :

Member Functions :

```
CVPCLHostTrans();
```

Financial Transaction Class (CVPCLFinancialTrans)

This is an abstract base class derived from the CVPCLHostTrans. This class is used to derive transaction classes which require a payment instrument (e.g., a Credit Card) associated with them to perform the transaction.

Class Name :

CVPCLFinancialTrans

Data :

Transaction Amount (CVPCLAmt)

Purchase Order Number (char[]])
Transaction Number (char[])
Authorization Identification Number (char[])
Retrieval Reference Number (char[])
Batch (CVPCLBatch)
Accumulators (CVPCLAccumulators)

Member Functions :

CVPCLFinancialTrans();
EStatus GetTransAmt(CVPCLAmt&);
EStatus GetPurchOrderNum(char *);
EStatus GetTransRefNum(char *);
EStatus GetRetRefNum(char *);
EStatus GetAuthId(char *);
EStatus GetCurrencyType(EPCLCurrency *);
EStatus SetPurchOrderNum(const char *);
EStatus SetTransRefNum(const char *);
EStatus SetRetRefNum(const char *);
EStatus SetAuthId(const char *);
EStatus SetCurrencyType (const char *)

Financial Credit Card Transaction Class (CVPCLFinCCTrans)

This is the base abstract class for the financial host transaction which require a Credit Card payment instrument. This class is derived from the CVPCLFinancialTrans.

Class Name :

CVPCLFinCCTrans

Data :

Credit Card Payment Instrument (CPCLCreditCard)

Member Functions :

CVPCLFinCCTrans();

Credit Card Authorization Only Transaction Class (CVPCL_CCAuthOnly)

This is the class derived from the CVPCLFinCCTrans class and implements the Authorization Only Transaction.

Class Name :

CVPCL_CCAuthOnly

Data :

Member Functions :

CVPCL_CCAuthOnly();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Credit Card Authorization with Capture Transaction Class (CVPCL_CCAuthCapt)

This is the class derived from the CVPCLFinCCTrans class and implements the Authorization with Data Capture Transaction.

Class Name :

CVPCL_CCAuthCapt

Data :

Member Functions :

CVPCL_CCAuthCapt();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Credit Card Return Transaction Class (CVPCL_CCReturn)

This is the class derived from the CVPCLFinCCTrans class and implements the Return Transaction.

Class Name :

CVPCL_CCReturn

Data :

Member Functions :

CVPCL_CCReturn();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Credit Card Pre-Authorization Transaction Class (CVPCL_CCPreAuth)

This is the class derived from the CVPCLFinCCTrans class and implements the Pre-Authorization Transaction.

Class Name :

CVPCL_CCPreAuth

Data :

Member Functions :

CVPCL_CCPreAuth();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

EStatus FormBatchRec();

Credit Card Off-line Authorization Only Transaction Class (CVPCL_CCOfflineAuth)

This is the class derived from the CVPCLFinCCTrans class and implements the Offline Authorization Class Transaction.

Class Name :

CVPCL_CCOfflineAuth

Data :

Member Functions :

CVPCL_CCOfflineAuth();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

EStatus FormBatchRec();

Credit Card Adjust Transaction Class (CVPCL_CCAdjust)

This is the class derived from the CVPCLFinCCTrans class and implements the Adjust Transaction.

Class Name :

CVPCL_CCAdjust

Data :

Member Functions :

CVPCL_CCAdjust();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Credit Card Void Transaction Class (CVPCL_CCVoid)

This is the class derived from the CVPCLFinCCTrans class and implements the Void Transaction.

Class Name :

CVPCL_CCVoid

Data :

Member Functions :

CVPCL_CCVoid();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Credit Card Forced Post Transaction Class (CVPCL_CCForcedPost)

This is the class derived from the CVPCLFinCCTrans class and implements the Forced Post Transaction.

Class Name :

CVPCL_CCForcedPost

Data :

Member Functions :

CVPCL_CCForcedPost();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Pre-Authorization Complete Transaction Class (CVPCL_CCPreAuthComp)

This is the class derived from the CVPCLFinCCTrans class and implements the Pre-Authorization Completion Transaction.

Class Name :

CVPCL_CCPreAuthComp

Data :

Member Functions :

CVPCL_CCPreAuthComp();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();
EStatus FormBatchRec();

Credit Card Balance Inquiry Class (CVPCL_CCBalanceInq)

This class is derived from the CVPCLFinCCTrans class and is used to perform the Merchant Balance Inquiry function.

Class Name :

CVPCL_CCBalanceInq

Data :

Member Functions :

CVPCL_CCBalanceInq();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Administrative Host Transaction Class (CVPCLAdminHostTrans)

This is an abstract base class derived from the CVPCLHostTrans class and is used to derive the administrative host transaction classes.

Class Name :

CVPCLAdminHostTrans

Data :

Member Functions :

CVPCLAdminHostTrans();

int GetHostIndex();

EStatus SetHostIndex (const int);

Reconcile Transaction Class (CVPCLReconcile)

This is the class derived from the CVPCLAdminHostTrans class and implements the Reconcile or Close functionality.

Class Name :

CVPCLReconcile

Data :

Member Functions :

CVPCLReconcile();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Host Log-on Transaction Class (CVPCLHostLogon)

This is the class derived from the CVPCLAdminHostTrans class and implements the Host Log-on Transaction.

Class Name :

CVPCLHostLogon

Data :

Member Functions :

CVPCLHostLogon();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Parameters Download Transaction Class (CVPCLParamsDwnld)

This is the class derived from the CVPCLAdminHostTrans class and implements the Parameters Download (VPOS configuration information from the host) functionality.

Class Name :

CVPCLParamsDwnld

Data :

Member Functions :

CVPCLParamsDwnld();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Test Transaction Class (CVPCLTestHost)

This is the class derived from the CVPCLAdminHostTrans class and implements the Test functionality which is used to test the host and the link.

Class Name :

CVPCLTestHost

Data :

Member Functions :

CVPCLTestHost();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Local Transaction Class Definitions (CVPCLLocalTrans)

This is the abstract base class for all the transactions that are performed locally to the VPOS.

Class Name :

CVPCLLocalTrans

Data :

Record Number (int)

Host Index (int)

Member Functions :

CVPCLocalTrans();

int GetRecNum();

int GetHostIndex()

EStatus SetRecNum(const int);

EStatus SetHostIndex(const int);

Virtual POS Lock/Stop Class (CVPCLVPosLock)

This class implements the VPOS Lock or the Stop Local functionality. Under the locked state the VPOS does not accept any transaction requests. The class is derived from the CVPCLLocalTrans base class.

Class Name :

CVPCLVPosLock

Data :

Member Functions :

CVPCLVPosLock();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Virtual POS UnLock/Start Class (CVPCLVPosUnlock)

This class implements the VPOS UnLock or the Start Local functionality. The class is derived from the CVPCLLocalTrans base class.

Class Name :

CVPCLVPosUnLock

Data :

Member Functions :

CVPCLVPosUnlock();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();

Transaction Data Administration Class (CVPCLTransDataAdmin)

This is an abstract base class used to derive the classes which are required to review/manage the transaction data which includes the batch data and the accumulator data. The class is derived from the CVPCLLocalTrans base class.

Class Name :

CVPCLTransDataAdmin

Data :

Member Functions :

CVPCLTransDataAdmin();

Batch Review Class (CVPCLBatchReview)

This class is derived from the CVPCLTransDataAdmin base class and implements the batch review functionality

Class Name :

CVPCLBatchReview

Data :

Member Functions :

CVPCLBatchReview();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Clear Batch Class (CVPCLClearBatch)

This class is derived from the CVPCLTransDataAdmin base class and implements the clear batch functionality, which is used to clear the batch in the event of doing a manual reconciliation between the VPOS and the acquirer.

Class Name :

CVPCLClearBatch

Data :

Member Functions :

CVPCLClearBatch();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Accumulators Review Class (CVPCLAccumReview)

This class is derived from the CVPCLTransDataAdmin base class and implements the Accumulators Review functionality.

Class Name :

CVPCLAccumReview

Data :

Member Functions :

CVPCLAccumReview();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();

Clear Accumulators Class (CVPCLClearAccum)

This class is derived from the CVPCLTransDataAdmin base class and implements the Accumulators Clear functionality.

Class Name :

CVPCLClearAccum

Data :

Member Functions :

CVPCLClearAccum();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();

VPOS Configuration Data Administration Class (CVPCLConfigDataAdmin)

This is an abstract base class and is used to derive classes which implement the functionality for managing the VPOS configuration data. The class is derived from the CVPCLLocalTrans base class.

Class Name :

CVPCLConfigDataAdmin

Data :

Member Functions :

Acquirer Data or the Host Definition Table Review Class (CVPCL_HDTReview)

This class is derived from the CVPCLConfigDataAdmin class and implements the Host Definition Table Review functionality.

Class Name :

CVPCL_HDTReview

Data :

Member Functions :

CVPCL_HDTReview();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Issuer Data or the Card Definition Table Review Class (CVPCL_CDTReview)

This class is derived from the CVPCLConfigDataAdmin class and implements the Card Definition Table Review functionality.

Class Name :

CVPCL_CDTReview

Data :

Member Functions :

CVPCL_CDTReview();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Communication Parameters Table Review Class (CVPCL_CPTReview)

This class is derived from the CVPCLConfigDataAdmin class and implements the Communications Parameters Table Review functionality.

Class Name :

CVPCL_CPTReview

Data :

Member Functions :

CVPCL_CPTReview();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Terminal Configuration Table Review Class (CVPCL_TCTReview)

This class is derived from the CVPCLConfigDataAdmin class and implements the Terminal Configuration Table Review functionality.

Class Name :

CVPCL_TCTReview

Data :

Member Functions :

CVPCL_TCTReview();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Acquirer Data or the Host Definition Table Update Class (CVPCL_HDTUpdate)

This class is derived from the CVPCLConfigDataAdmin class and implements the Host Definition Table Update functionality.

Class Name :

CVPCL_HDTUpdate

Data :

Member Functions :

CVPCL_HDTUpdate();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Issuer Data or the Card Definition Table Update Class (CVPCL_CDTUpdate)

This class is derived from the CVPCLConfigDataAdmin class and implements the Card Definition Table Update functionality.

Class Name :

CVPCL_CDTUpdate

Data :

Member Functions :

CVPCL_CDTUpdate();

EStatus InitializeTrans(TVPosParamsBlk *);

EStatus ExecuteTrans(TVPosResultsBlk *);

EStatus ShutDownTrans();

Communications Parameters Table Update Class (CVPCL_CPTUpdate)

This class is derived from the CVPCLConfigDataAdmin class and implements the Communications Parameters Table Update functionality.

Class Name :

CVPCL_CPTUpdate

Data :

Member Functions :

CVPCL_CPTUpdate();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();

Terminal Configuration Table Update Class (CVPCL_TCTUpdate)

This class is derived from the CVPCLConfigDataAdmin class and implements the Terminal Configuration Table Update functionality.

Class Name :

CVPCL_TCTUpdate

Data :

Member Functions :

CVPCL_TCTUpdate();
EStatus InitializeTrans(TVPosParamsBlk *);
EStatus ExecuteTrans(TVPosResultsBlk *);
EStatus ShutDownTrans();

Batch Class (CVPCLBatch)

This class defines the batch record and the operations which are performed on the batch.

Class Name :

CVPCLBatch

Data :

Batch Record Structure (TVPosBatchRec)

```
// Definition of the TVPosBatchRec is as below,
typedef struct _VPosBatchRec
{
        char  szTransAmt[];
        char  szTransDate[];
        char  szTransTime[];
        char  szRetrievalRefNum[];          // Trans. Ref. No. sent by
the host
        char  szAuthId[];                   // Approval Code sent by the
host
        char  szOrigAmt[];                  // Original amount for -
Adjust
        char  szPurchOrderNum[];
        char   szBatchNum[];
        EPCLTransType  TransType;
        EPCLPmtInst    PmtInst;
        EPCLCurrency   CurrencyType;
        EPCLDecimals   NumDecDigits;
        unsigned int   nTransRefNum;     // Running Ref. Number gen.
by the                                          //VPOS for every
approved txn.          unsigned long  lnSTAN;          // Sys.
Trace Number incr. by VPOS
                                                // for every trans. that is
trans. to host
        TPmtInstData   PayInstData;
} TVPosBatchRec;
```

Member Functions :

CVPCLBatch();

```
EStatus SetTransType(const EPCLTransType);
EStatus SetRetRefNum(const char *);
EStatus SetAuthId(const char *);
EStatus SetPurchOrderNum(const char *);
EStatus SetTransRefNum(const long);
EStatus SetTransAmt(const char *);
EStatus SetBatchNum(const char *);
EStatus SetSTAN(const long);
EStatus SetDateMMDDYYYY(const char *);
EStatus SetTimeHHMMSS(const char *);
EStatus SetPmtInst(const EPCLPmtInst);
EStatus SetCCAcctNum(const char *);
EStatus SetCCExpDate(const char *);
EStatus SetOrigAmt(const char *);
EStatus GetBatchRec(TVPosBatchRec *);
EStatus InitBatch();
EStatus OpenBatch(const char *, FILE **, const char *);
EStatus CloseBatch(FILE *);
EStatus AddBatchRec ();                    // Adds a record to the
batch
EStatus GetBatchRec (const long);          // Gets a record from the
batch
EStatus UpdateBatchRec (const long);       // Update batch record
with NR
EStatus DeleteBatchRec (const long);       // Deletes the batch record
```

Accumulator Class (CVPCLAccum)

This class defines the Accumulator record and the operations on the accumulators.

Class Name :

CVPCLAccum

Data :

Credit Amount (char szCreditAmt[AMT_SZ + 1])

Credit Count (int nCreditCnt)

Debit Amount (char szDebitAmt[AMT_SZ + 1)

Debit Count (int nDebitCnt)

Member Functions :

int OpenAccum(int fHandle);

int GetAccum (int nAccumType, int *pnAccumCnt, char *pszAccumAmt);

int CloseAccum(int fHandle);

int CleanAccum();

Host Definition Table Class (CVPCL_HDT)

This class defines the Host Definition Table record and the operations on the table.

Class Name :

CVPCL_HDT

Data :

Host Definition Table Record Structure (TVPosHDTRec)

The TVPosHDTRec structure contains the following fields,

```
typedef struct _VPosHDTRec
{
        char  szTermId[];
        char  szMerchId[];
        char  szBatchNum[];
        char  szTPDU[];
        char  szNII[];
```

```
        char  szHostName[];
        EPCLHostProtType HostProtType;
        EPCLHostProtSubType  HostProtSubType;
        // Data Capture Required Flags
        VPosBool fAuthOnlyDC;
        VPosBool fAuthCaptDC;
        VPosBool fForcedPostDC;
        VPosBool fAdjustDC;
        VPosBool fReturnDC;
        VPosBool fOfflineAuthDC;
        VPosBool fVoidDC;
        VPosBool fPreAuthDC;
        VPosBool fPreAuthCompDC;
        unsigned int  nNumAdv;// Max. No. of piggy-back trans.
allowed
        unsigned int  nTransRefNum;
        unsigned long     lnSTAN;     // Systems Trace Number
    } TVPosHDTRec;
```

Member Functions :

```
    CVPCL_HDT();
    EStatus CleanHDT();
    EStatus LoadHDTRec(const int);
    EStatus SaveHDTRec(const int);
    EStatus GetNumRecs(int *);
    EStatus GetHDTRec(TVPosHDTRec *);
    EStatus GetTermId(char *);
    EStatus GetMerchId(char *);
    EStatus GetBatchNum(char *);
```

```
EStatus GetTransRefNum(unsigned int *);
EStatus GetTPDU(char *);
EStatus GetNII(char *);
EStatus GetHostName(char *);
EStatus GetHostProtType(EPCLHostProtType *);
EStatus GetHostProtSubType(EPCLHostProtSubType *);
EStatus GetNumAdv(unsigned int *);
EStatus GetSTAN(unsigned long *);
EStatus GetAuthOnlyDC(VPosBool *);
EStatus GetAuthCaptDC(VPosBool *);
EStatus GetAdjustDC(VPosBool *);
EStatus GetReturnDC(VPosBool *);
EStatus GetForcedPostDC(VPosBool *);
EStatus GetOfflineAuthDC(VPosBool *);
EStatus GetVoidDC(VPosBool *);
EStatus GetPreAuthDC(VPosBool *);
EStatus GetPreAuthCompDC(VPosBool *);
EStatus SetHDTRec(TVPosHDTRec *);
EStatus SetTermId(const char *);
EStatus SetMerchId(const char *);
EStatus SetBatchNum(const char *);
EStatus SetTransRefNum(const unsigned int);
EStatus SetTPDU(const char *);
EStatus SetSTAN(const unsigned long);
EStatus SetNII(const char *);
EStatus SetHostName(const char *);
EStatus SetHostProtType(const EPCLHostProtType);
EStatus SetHostProtSubType(const EPCLHostProtSubType);
EStatus SetNumAdv(const int);
```

```
EStatus SetAuthOnlyDC(const VPosBool);
EStatus SetAuthCaptDC(const VPosBool);
EStatus SetAdjustDC(const VPosBool);
EStatus SetReturnDC(const VPosBool);
EStatus SetForcedPostDC(const VPosBool);
EStatus SetOfflineAuthDC(const VPosBool);
EStatus SetVoidDC(const VPosBool);
EStatus SetPreAuthDC(const VPosBool);
EStatus SetPreAuthCompDC(const VPosBool);
```

Card Definition Table Class (CVPCL_CDT)

This class defines the Card Definition Table record and the operations on the table.

Class Name :

CVPCL_CDT

Data :

Card Definition Table Record Structure (TVPosCDTRec)

The TVPosCDTRec structure contains the following fields,

```
typedef struct _VPosCDTRec
{
        char  szPANLo[];
        char  szPANHi[];
        char  szCardLabel[];
        int   nHostIndex;
        int   nMinPANDigit;
        int   nMaxPANDigit;
        // Transaction Allowed Flags
        VPosBool fAuthOnlyAllwd;
        VPosBool fAuthCaptAllwd;
```

```
        VPosBool fForcedPostAllwd;
        VPosBool fAdjustAllwd;
        VPosBool fReturnAllwd;
        VPosBool fOfflineAuthAllwd;
        VPosBool fVoidAllwd;
        VPosBool fPreAuthAllwd;
        VPosBool fPreAuthCompAllwd;
} TVPosCDTRec;
```

Member Functions :

```
CVPCL_CDT();
EStatus CleanCDT();
EStatus LoadCDTRec(const int);
EStatus SaveCDTRec(const int);
EStatus GetNumRecs(int *);
EStatus GetCDTRec(TVPosCDTRec *);
EStatus GetPANLo(char *);
EStatus GetPANHi(char *);
EStatus GetCardLabel(char *);
EStatus GetCDTHostIndex(int *);
EStatus GetMinPANDigit(int *);
EStatus GetMaxPANDigit(int *);
EStatus GetAuthOnlyAllwd(VPosBool *);
EStatus GetAuthCaptAllwd(VPosBool *);
EStatus GetAdjustAllwd(VPosBool *);
EStatus GetReturnAllwd(VPosBool *);
EStatus GetOfflineAuthAllwd(VPosBool *);
EStatus GetVoidAllwd(VPosBool *);
EStatus GetPreAuthAllwd(VPosBool *);
```

EStatus GetPreAuthCompAllwd(VPosBool *);
EStatus GetForcedPostAllwd(VPosBool *);
EStatus SetCDTRec(TVPosCDTRec *);
EStatus SetHostIndex(const int);
EStatus SetMinPANDigit(const int);
EStatus SetMaxPANDigit(const int);
EStatus SetPANLo(const char *);
EStatus SetPANHi(const char *);
EStatus SetCardLabel(const char *);
EStatus SetAuthOnlyAllwd(const VPosBool);
EStatus SetAuthCaptAllwd(const VPosBool);
EStatus SetAdjustAllwd(const VPosBool);
EStatus SetReturnAllwd(const VPosBool);
EStatus SetForcedPostAllwd(const VPosBool);
EStatus SetOfflineAuthAllwd(const VPosBool);
EStatus SetVoidAllwd(const VPosBool);
EStatus SetPreAuthAllwd(const VPosBool);
EStatus SetPreAuthCompAllwd(const VPosBool);

Communications Parameters Table Class (CVPCL_CPT)

This class defines the communications parameters table and the operations on the table.

Class Name :

CVPCL_CPT

Data :

Communications Parameters Table Record Structure (TVPosCPTRec)

The TVPosCPTRec structure contains the following fields,

typedef struct _VPosCPTRec

```
{
        char  szAcqPriAddress[];
        char  szAcqSecAddress[];
        char  szAcqTerAddress[];

        int   nRespTimeOut;
} TVPosCPTRec;
```

Member Functions :

```
CVPCL_CPT();
EStatus CleanCPT();
EStatus LoadCPTRec(const int);
EStatus SaveCPTRec(const int);
EStatus GetNumRecs(int *);
EStatus GetCPTRec(TVPosCPTRec *);
EStatus GetAcqPriAddress(char *);
EStatus GetAcqSecAddress(char *);
EStatus GetAcqTerAddress(char *);
EStatus GetRespTimeOut(int *);
EStatus SetCPTRec(TVPosCPTRec *);
EStatus SetAcqPriAddress(const char *);
EStatus SetAcqSecAddress(const char *);
EStatus SetAcqTerAddress(const char *);
EStatus SetRespTimeOut(const int);
```

Terminal Configuration Table Class (CVPCL_TCT)

This class defines the VPOS terminal configuration parameters table and the operations on the table.

Class Name :

CVPCL_TCT

Data :

Terminal Configuration Table Record Structure (TVPosTCTRec)

The TVPosTCTRec structure contains the following fields,

```
typedef struct _VPosTCTRec
{
  char  szMerchName[];
  VPosBool fVPosLock;                    // VPOS Lock/Unlock Toggle
Flag
} TVPosTCTRec;
```

Member Functions :

```
CVPCL_TCT();
EStatus LoadTCTRec();
EStatus SaveTCTRec();
EStatus CleanTCT();
EStatus GetTCTRec(TVPosTCTRec *);
EStatus GetMerchName(char *);
EStatus GetVPOSLock(VPosBool *);
EStatus SetMerchName(const char *);
EStatus SetVPOSLock(const VPosBool);
```

Amount Class (CVPCLAmount)

This class defines the amount data items and the operations on them.

Class Name :

CVPCLAmount

Data :

Amount (char[])

Currency Type (EPCLCurrency)

Member Functions :

CVPCLAmount();

EStatus Initialize(const CPCLAmount&);

EStatus Initialize(const char *);

EStatus Initialize(const long);

void operator = (const char *);

void operator = (const long);

EStatus GetAmount(char *);

operator const char * () const;

operator const long ();

Payment Instruments Class (CPCLPmtInst)

This section defines the Payment Instrument Class hierarchy. Figure **16** illustrates a transaction class hierarchy in accordance with a preferred embodiment.

Class Name :

CPCLPmtInst

Data :

Payment Instrument Type (EPCLPmtInst)

Member Functions :

CPCLPmtInst();

EStatus GetPmtInstType(EPCLPmtInst *);

Bank Cards Class (CPCLBankCard)

This class is derived from the CPCLPmtInst class and implements the bank cards class.

Class Name :

CPCLBankCard

Data :

Account Number (char[ ])

Expiration Date (CPCLDateTime)

Index into the CDT table (int)

Member Functions :

CPCLBankCard();

EStatus Initialize();

EStatus SetAcctNum(const char *);

EStatus SetExpDate(const char *);

EStatus GetAcctNum(char *);

EStatus GetExpDate(char *);

EStatus ValidateCard();

int GetCDTIndex();

VPosBool DoLuhnCheck();

VPosBool DoCardRanging();

EStatus DoValidateExpDate();

Credit Cards Class (CPCLCreditCard)

This class is derived from the CPCLBankCard class and has the same data and the methods as the CPCLBankCard class.

Class Name :

CPCLCreditCard

Data :

Member Functions :

CPCLCreditCard();

Debit Cards Class (CPCLDebitCard)

This class is derived from the CVPCLBankCard class and implements the debit card class.

Class Name :

CPCLDebitCard

Data :

Card Holder Encrypted PIN (char[ ])

Member Functions :

CPCLDebitCard();

EStatus GetEncryptedPIN(char *);

EStatus SetEncryptedPIN(char *);

VPOS Class Library Interface and API Definition

This section explains the classes which provide the interface to the VPOS class library.

Data Structures required for the VPOS Interface Class

Transaction Parameters Structure (TVPosParamsBlk) - This structure is a subset of all the transaction parameters required for the different transactions.

```
typedef struct _VPosParamsBlk
{
        char szTransAmt[];      // Without decimal point.
```

```
                                    // Left most two digits implied to be decimal
digits
        char szPurchOrderNum[];
                char szRetRefNum[];
        char szBatchNum[];
                char szNewBatchNum[];
                char szOrigAmt[];
                char szCPSData[] ;
                char szAuthId[];        // Auth Id for offline auth-only
transaction
                int HostIndex;
                unsigned int nTransRefNum;
                VPosBool fVPosLock;
                ECPSDataType eCPSType ;
                EPCLTransType TransType;
                EStatus TransResult;
                EPCLPmtInst PmtInst;
                EPCLCurrency CurrencyType;
                EPCLDecimals NumDecDigits;
                EVPCLAccumType AccumType;
                TPmtInstData PayInstData;
                union _VPosConfigData
                {
                        TVPosHDTRec srHDTRec;
                        TVPosCDTRec srCDTRec;
                        TVPosCPTRec srCPTRec;
                        TVPosTCTRec srTCTRec;
                } VPosConfigData;
                void *Context;              // Context from the calling interface
```

```
        EStatus (*VPosCallBack)(TVPosResultsBlk *, void *);

    } TVPosParamsBlk;
```

Transaction Results Structure (TVPosResultsBlk) - This structure contains all the fields returned from the host and other fields which are required for doing terminal data capture.

```
    typedef struct _VPosResultsBlk
    {
            char szNewBatchNum[];
            int  nHostIndex;
            EStatus TransResult;
            TVPosBatchRec srBatchRec;
            TVPosAccumRec srAccumRec;
            char szCardLabel[];
            TVPosHDTRec srHDTRec;
            TVPosCDTRec srCDTRec;
            TVPosCPTRec srCPTRec;
            TVPosTCTRec srTCTRec;
    } TVPosResultsBlk;
```

The various status codes for the enumeration EStatus are detailed below.

VPOS Interface Class (CVPosInterface)

This class provides the interface to the VPOS Transaction Class Library.

Class Name :

CVPosInterface

Data :

Member Functions :

```
CVPosInterface();
// Creates the Transaction Object, takes care
// of other initialization and executes the transaction.
CVPCLTransaction *pclTransFactory(TVPosParamsBlk *);
EStatus DestroyTrans(CVPCLTransaction *);
```

VPOS API Definition

This section explains in the VPOS API which are required for interfacing with the VPOS Class Library. All the different VPOS transactions can be initiated using the API defined in this section.

VPosInitialize - Initialize VPOS

This API is used to start and initialize the VPOS. The API definition is disclosed below.

API Definition :

VPosBool VPosInitialize(void);

Parameters :

None

Returns :

TRUE or FALSE indicating whether the function call was a success.

VPosExecute - Execute a VPOS Transaction

This API is used to execute a particular VPOS transaction.

API Definition :

VPosBool VPosExecute(TVPosParamsBlk *, TVPosResultsBlk *)

Parameters :

Pointer to the Parameters Structure (TVPosParamsBlk)

Pointer to the Results Structure (TVPosResultsBlk)

Returns :

TRUE or FALSE indicating whether the function call was a success.

VPosShutDown - Shutdown the VPOS

This is used to shutdown the VPOS.

API Definition :

VPosBool VPosShutDown(void)

Parameters :

None

Returns :

TRUE or FALSE indicating whether the function call was a success.

VPOS Status Codes

This section details the different status codes (listed under the enumeration EStatus) which the VPOS returns for the different operations performed.

```
enum EStatus
{
    eSuccess = 0,               // Function call or operation successful
    eFailure,                   // General failure
    eVPosLocked,                // Vpos locked, transaction not allowed
                                // Transaction related error codes
    ePmtInstNotSupported,       // Payment Instrument not supported
    eTransNotSupported,         // Transaction type not supported
    eTransInitErr,              // Transaction Initialization Failed
    eAdjustNotAllwd,            // Adjust not allowed on this
transaction
    eVoidNotAllwd,              // Void not allowed on this transaction
```

```
        eForcedPostNotAllwd,        // Forced Post not allowed on this
transaction
        ePreAuthCompNotAllwd,       // Pre-Auth. not allowed on this
transaction
        eAmtErr,                    // Error in the amount passed
        eHDTLoadErr,                // Error during loading the HDT table
        eCDTLoadErr,                // Error during loading the CDT table
        eCPTLoadErr,                // Error during loading the CPT
table
        eTCTLoadErr,                // Error during loading the TCT
table
        eHDTWriteErr,               // Error during writing to the HDT
table
        eCDTWriteErr,               // Error during writing to the CDT
table
        eCPTWriteErr,               // Error during writing to the
CPT table
        eTCTWriteErr,               // Error during writing to the
TCT table
        eTCTFieldErr,               // Error handling a TCT table
field
        eLuhnErr,                   // Luhn check failed on the account
        eRangingErr,                // Card range not found
        ePANLenErr,                 // PAN length error
        eExpiredCard,               // Card expired
        eInvalidMonth,              // Invalid month in the expiration date
        eFileOpenErr,               // General file open error
        eFileCloseErr,              // General file close error
```

VPOS Terminal Architecture

Figure **25** is a block diagram of the vPOS Terminal Architecture in accordance with a preferred embodiment. The Internet **2500** provides the communication processing necessary to enable the VPOS Terninal architecture. The terminal interface CGI **2520** communicates via the internet to provide information to the VPOS OLE Server **2550** which formats information in accordance with the VPOS API DLL **2560** which uses the protocol class DLL **2570** to flesh out the message for delivery to the Gateway Server **2580**. The collection of the VPOS OLE Server **2550**, VPOS API DLL **2560** and the Protocol Class DLL **2570** make up the VPOS Software Development ToolKit (SDK) which are used to enable VPOS applications for interfacing with an Operator **2540**.

VPOS/VGATE Architecture

The architecture of the Virtual Point of Sale (VPOS) and Virtual Gateway (VGATE) architecture maintains SET compliance while providing support for additional message types that are not enabled in SET. The architecture includes isolation of cryptographic details in a single module to facilitate single version government approval while maximizing the flexibility of the system for customization and facilitating transfer of updated versions on an acquirer specific basis. Figure **18** is a block diagram of the extended SET architecture in accordance with a preferred embodiment. Processing commences at function block **1800** for a consumer-originated transaction via the World Wide Web (WWW) or **1810** for a merchant-originated transaction on the internet. In either case control passes immediately to the WWW server **1820** for the transaction to be appropriately formatted and the appropriate interface page presented, whether the transaction is a store front **1822**, shopping cart **1824**, pay page **1826**, standard terminal administration **1828-1830** transaction, or an extended terminal transaction

1834. If processing requires authentication of the transaction, then control passes through the Virtual Point of Sale (VPOS) Application Programming Interface (API) library **1840** for SET compliant transactions and through the VPOS API extensions library for extensions to the SET protocol. Then, at function block **1842**, if the transaction is SET compliant, and function block **1864** if the transaction is not SET compliant, a library of protocol stack information is used to conform the message before it is transmitted to a Gateway site for ultimate delivery to a bank host **1874** for authorization.

Extended SET messages are processed at the Gateway site on a two track basis with the division criteria being SET compliance (which will change over time as more functionality is put into SET) or SET extensions. Set compliant messages are processed via the protocol statck library **1862**, while SET extensions are processed via the protocol stack entension library **1864**. Then, at function block **1870** the gateway engine processes SET and Host specific code including gateway administration extensions **1872** that bypass the normal processing and flow directly from the merchant and consumer server **1820** to the gateway administration extensions **1872** to the Gateway Engine **1870**.

As described above, there are three channels by which messages are exchanged between VPOS **1846** and VGATE **1856**.

1. Standard SET messages

The standard SET messages are originated by the merchant software either via a pay page **1826** directly controlled by the consumer, or via an operator interface consisting of a set of HTML pages and associated executables launched by the pages (e.g. pay page **1826** and standard terminal administration **1828-1830**.)

Each SET message type (e.g., authorization v. capture) transmits a different set of data and each requires a different Protocol Data Unit (PDU) to describe its encoding. Examples of how Standard SET messages are encoded are given in the SET documentation previously incorporated by reference.

2. Extended SET messages

The Extended SET messages are utilized as an "escape mechanism" to implement acquirer-specific messages such as settlement/reconciliation, employee logon/logoff, and parameter download. The messages are developed as a set of name-value pairs encapsulated in a PKCS-7 wrapper and wrapped in Multipurpose Internet Mail Extensions (MIME), described in a book by N. Borenstein & N. Freed, "RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies" (Sep. 1993). The name-value pairs can have arbitrary (8-bit) data, so arbitrary items can be passed through the extended SET channel, including executable programs and Dynamic Load Libraries (DLL)s.

Figure **18B** illustrates a multipart MIME message with one Extended SET message and one Standard SET authorizing message. Mime is utilized as an outer wrapper **1890** to allow an Extended SET message **1891** to be transmitted as a compon of messages embedded in one MIME multipart message. In this manner, a standard SET message can be sent with an Extended SET message in one VPOS/VGATE communication transaction.

Embedding the Extended SET messages in a PKCS-7 wrapper enables the same message authentication to occur as in standard SET messages. Thus,

for SET-compliant and non-SET-compliant messages, the same mechanism may be
used to restrict which entities the vPOS or vGATE will trust in any communications. An important concept in Extended SET is that all messages, of any type, are sent in a uniform name/value pair format, thus allowing a single Protocol Data Unit to suffice for any type of message sent through the Extended SET channel. Since arbitrary data may be sent this way, a mechanism must be provided to preclude the use of the Extended SET channel by parties other than approved financial institutions. If this is not ensured, then the NSA and the US Department of Commerce will not approve the software for export.

SET itself to some degree ensures that this Extended SET channel is used only by financial institutions. The
protocol stack extension library only processes messages that have been signed by a financial institution SET certificate that is in turn signed by a payment instrument brand certificate (such as Visa or MasterCard). Stronger control over the Extended SET channel can be achieved by further restricting processing of messages to those signed (either instead of or in addtion to the financial institution SET certificate) by a second certificate belonging to a third-party agency, either governmental or private (e.g., VeriFone, as manufacturer of the software).

In this way, a particular set of Extended SET messages can be implemented by Bank X, and a different set of messages by Bank Y. If a vPOS has an extended terminal transaction interface as shown in Figure **18A** at block **1834** for Bank X, and has been configured to only accept messages from a vGate with Bank X's certificate, then it will be able to communicate those messages to a vGate that has the certificate for Bank X, and accepts

messages of the types in Bank X's message set. The vPOS will not be able to connect to the Bank Y gateway, or to any other system that purports to communicate via Extended SET. This restriction is further secured by utilizing a public key certificate that is "hard wired" into vPOS, and which is distributed only to gateways that use the Extended SET mechanism and which have been approved for export by the US Commerce Department.

Figure **18C** is an example flowchart of message processing in accordance with a preferred embodiment. Processing commences at function block **1880** when a message is received by an HTTPS server or other listener and passed to decision block **1883** to determine if the sending VPOS has transmitted an authentic message and if the VPOS is authorized to communicate with this gateway. If the message is not authentic, then the message is logged as an error and the error is handled as shown in function block **1889.** If the message is authentic, then the message is decrypted at function block **1884** and the PDU parses the message into name / value pairs. Then, based on the message type and the extended SET version information, the remaining message is parsed at function block **1885** and the message is checked for conformance to the appropriate specification as shown at decision block **1887.** If the message does not conform, then it is logged and the error handled at function block **1889.** If the message conforms to the proper specification in decision block **1887** then the message is translated into the appropriate host format and sent to the host as shown in function block **1888.** Thus, when a gateway receives an incoming message from a vPOS and parses the Extended SET portion of the message, a single MIME message can transmit a SET message and/or an Extended Set Message.

An export license for the encrption can be obtained on a case-by-case basis, and since there will be potentially millions of VPOS's, it is desireable to obtain a commodities jurisdiction for the vPOS, to enable a single version of the VPOS (rather than one version for each bank) to be supported by the VPOS architecture. The architecture described here ensures that the single version of VPOS, no matter how it is configured with extended terminal transaction
interfaces, cannot be used to communicate any data other than that contained in the extended SET messages that have been approved for export by the US government to be used exclusively for a specific bank.

Figure **18D** is an example of a simple message between vPOS and vGate using the Extended SET channel enabling an employee to sign on, or "logon" to a given terminal in accordance with the subject invention. The message must contain the employee's logon ID, a password to be verified by the bank host computer, and the date and time as shown at **1894**.

While the contents of the message are shown without encryption in Figure **18D**, it should be noted that the information (including the logon password) are SET encrypted inside the PKCS-7 wrapper **1894**. Certain fields may be designated as mandatory for an Extended SET message, to allow the vGate or vPOS to decide how to handle the message. For the sake of clarity, in this message **1894**, only two fields, "messagetype" and "ESETversion", are mandatory. These fields inform the vGate that this message is of type "logon," and that the vPOS is using version "1.0A" of the ESET message formats defined for the vGate. In this embodiment, the length indicator "[5]" is used to distinguish the length (in bytes) of the field of type "messagetype" in the message. In this way, there are no special end-of-data characters, and therefore arbitrary data need not have any "escaped" characters.

It should be noted that using escaped characters will work equally well. Total message integrity is assured by the digital signatures in the PKCS-7 wrapper. This does not, however, preclude the use of other checksumming schemes for additional pinpointing of transmission or encoding errors. The messagetype and ESETversion name/value pairs facilitate vGate look up of what name/value pairs are expected in the "logon" message. Some name/value pairs may be mandatory, and others may be optional.

Figure **18E** is an example of a simple message between vPOS and vGate using the Extended SET channel enabling an employee to sign on, or "logon" to a given terminal in accordance with the subject invention. In response to the logon request message from a vPOS, the vGate may respond with a "logon accepted" message **1894**, as depicted in Figure **18E**, which vPOS, upon receipt and authentication, then uses to unlock the terminal for that user.

3. Gateway-intitiated messages

Since all SET messages between a merchant and an acquirer are currently merchant-initiated, there must be a separate mechanism for initiating a message from a gateway, for example to request the upload of MIB data, or to download new parameters. This is accomplished by requiring the gateway to send a message to the merchant via a MIME-encapsulated PKCS-7 message containing name-value pairs to the merchant server directly, rather than to the SET module. This channel is shown in Figure **18A** at block **1860.**

The message is verified for origination from the acquirer, and is utilized to either initialize a merchant action, such as to update the merchant's

administration page (for example by blinking a message saying, "PLEASE RE-INITIALIZE YOUR TERMINAL"), or by initiating a request/response message pair originating from the merchant (for example, "HERE ARE THE CONTENTS OF MY MIB"). This is achieved by calling one of the extended terminal transaction interfaces (Figure **18A** at **1834**), which in turn indicates a SET transaction.

Gateway Customization via the Extended SET Channel

Gateway customization in extended SET is extremely powerful and a novel concept for VPOS processing. Each VPOS contains a "serial number" which is unique to each copy of the software. Once a merchant has selected an acquirer and obtained the appropriate certificates, the VPOS can be customized utilizing the communication link and messages containing customization applications.

Let us consider an example in which a Wells Fargo Bank (WFB) distributes VPOS via different sales channels. The first is direct from WFB to an existing merchant with whom WFB already has an existing relationship. In this case, a version of VPOS customized for WFB is sent to the merchant. The customizations may involve modification or replacement of, for example, a store front **1822,** shopping cart **1824,** pay page **1826,** standard terminal administration transaction interface **1828-1830** or an extended terminal transaction interface **1834.**

Using the built-in "serial number" certificate and the Test Gateway public key, it is possible to securely download customization applications to a specific VPOS application. Once the VPOS is appropriately configured, the last stage of customization download is to configure the VPOS so that it only responds to a public key certificate of the merchant's acquirer.

Thread Safe VPOS - TID Allocation

Physical terminals process a single transaction at a time since clerks are usually only able to process one transaction at a time. Web Servers can process many transactions at a time, so payment requests can often occur simultaneously. Thus, the VPOS Software must have support for multi-tasking and provide support for multiple threads to be active at the same time in the same system as well as the same process. This requirement is relatively straight forward. However, the authorizing banks require that all transaction requests include a Terminal ID (TID), and, for many banks, no single TID may be active in any two transaction requests that overlap in time. Thus, the VPOS requires dynamic allocation of TIDs to requesting threads.

To provide for this requirement, the VPOS provides a TID pool in tabular form in a database management system (DBMS). This table has two colums: TID NAME & Allocation date/time. If the TID date is null, then the TID is not in use and may be assigned. A date/time field is utilized to allow TID allocations to expire. TID requests are made utilizing a SQL query on the TID Pool to find the first null or expired date/time, which is replaced with the current date/time and the TID name returned.

REMOTE VPOS

The unique archtitecture of the Cardholder **120**, Merchant **130** and Gateway **140,** as shown in Figure **1B**, provides communication capability between the modules utilizing the internet to support linkages **150** and **170**. Since the internet is so pervasive, and access is available from virtually any computer, utilizing the internet as the communication backbone for connecting the cardholder, merchant and access to the authorizing bank through a gateway

allows the merchant VPOS software to be remotely located from the merchant's premises. For example, the cardholder could pay for goods from any computer system attached to the Internet at any location in the world. Similarly, the merchant VPOS system could be located at a central host site where merchant VPOS systems for various merchants all resided on a single host with their separate access points to the Internet. The merchant could utilize any other computer attached to the Internet utilizing a SSL or SET protocol to query the remote VPOS system and obtain capture information, payment administration information, inventory control information, audit information and process customer satisfaction information. Thus, without having to incur the overhead of maintaining sufficient computer processing power to support the VPOS software, a merchant can obtain the information necessary to run a business smoothly and avoid hiring IS personnel to maintain the VPOS system.

VPOS Multi-Merchant Processing

Multiple merchant processing refers to the ability of a plurality of merchants to process their individual VPOS transactions securely on a single computer. The archtecture relies on each payment page obtaining the merchant name in a hidden field on the payment page. The VPOS engine receives the merchant name with a particular transaction and synchronizes the processing utilizing a Set Merchant method. This command causes the VPOS API to look up a unique registry tree based on the merchant name. This process causes the VPOS engine to engage the appropriate configuration to process the transaction at hand utilizing a Microsoft Registry Tree. A registry tree contains Card Definition Tables (CDT)s, Acquirer Definition Tables (ADT)s, Merchant Definition Tables (MDT)s, Protocol Configuration Tables (PCT)s, etc. The CDTs point to specific ADTs since each supported card can be supplied by a distinct acquirer. This is

one form of split connection. Each of the ADTs in turn point to PCTs, and some acquirers can support multiple parallel gateways. A merchant's name refers to a unique database in the database management system which contains for example, TIDs.

So, for example, to fully qualify a particular merchant in a multi-merchant system, the Acquirer Definition Table is queried to ascertain the particular Gateway (VFITest), then if Bank of America requires verification of network communication information, the particular CardDT is accessed with for example VISA. The particular merchant will service VISA transactions utilizing a particular acquirer. The particular piece of merchandise will also be detailed in a data base. Finally, the merchant Configurations wil l also be stored in the database to facilitate E-mail and name lookup.

VPOS CLIENT

The interaction between the VPOS and a client commences when a pay page solicits parameters of a transaction. Then, the parameters are validated to be sure the payment instrument, for example, cardnumber is not null. Then, a transaction object is created, eg. AUTHONLY, and the object is initialized and stuffed with parameters of the transaciion, eg. ao.setpan(accnum), and the object is executed. This execution invokes the VPOS engine. The VPOS engine further validates the parameters based on the particular merchant's configuration. For example, some merchans do not accept American Express Cards, but will take Visa, and all merchants check the expiration date of the card. Assuming a valid and acceptable card has been tendered, then a TID is assigned (expiring, existing TIDs) or block a new TID from the TID Pool. This generates a STAN, XID, RRPID unique tag and creates an initial record in the transaction database which is flagged as before gateway processing in case the transaction crashes and

must be backed out. Then the protocol parameters are identified in the registry based on card type, and a particular acquirer identified. Then, a protocol object is created and executed to extract results from the protocol object and the before gateway "bit" is flipped to again flag the location of the transaction in the process as it is submitted to the Gateway.

The results received back from the Gateway are placed into a transaction object with is reported back to the pay page and ultimatey back to the pay page user.

VPOS Merchant Pay Customization

A novel feature of the VPOS software provides payment page customization based on a merchant's preferences. This feature automatically lists cards that are accepted by a particular merchant based on the active terminal configuration. Each approved card for a particular merchant is linked to the display via an URL that provides a pointer to the credit card information supported by the merchant. Each card has an entry in a data structure referred to as the Credit Definition Table (CDT).

A preferred embodiment of the VPOS merchant pay customization software in accordance with a preferred embodiment is provided in Figure **19** which illustrates the logic utilizing a flowchart, and a listing of the source code below. Processing commences at terminal **1900** and immediately flows to function block **1910** where an index variable is initialized for stepping through each of the accepted payment instruments for the merchant's page. Then, at function block **1930**, a URL key is obtained associated with the current merchant pay page and index value. The URL key is a registry key name that points to a picture of a credit card that the merchant has associated with the pay page and which the merchant accepts as payment.

At output block **1940** the card image associated with the URL key is obtained and displayed on the terminal. The CDT entry is obtained at function block **1950** utilizing the URL key. The CDT is utilized for storing information associated with each card. Then, at decision block **1960**, a test is performed to determine if the last payment method card has been processed and displayed on the merchant display. If not, then the index is incremented at function block **1920** and the loop reiterated to process the next card at function block **1930**. If all the cards have been processed, then control is returned to the merchant program for processing the transaction at terminal **1970**.

Figure **20A-20H** are block diagrams and flowcharts setting forth the detailed logic of thread processing in accordance with a preferred embodiment. Figure **20A** illustrates a prior art approach to POS processing utilized in most grocery stores and department stores today. Figure **20B** is a data structure of a POS transaction request in accordance with a preferred embodiment. Figure **20C** illustrates VPOS architecture with account requests being processed by a single acquiring bank. Figure **20D** illustrates a VPOS Transaction request data structure in accordance with a preferred embodiment. The transaction type, VPOS Terminal ID, Amount, Currency type, Digital Signature, Account number, Expiration date and other information are all stored in this record for later retrieval. Figure **20E** is a blow up of a portion of a TID allocation database in accordance with a preferred embodiment. Figure **20F-H** are flowcharts of the detailed logic in accordance with a preferred embodiment.

```
#include "rr.h"
#ifndef _NT
#define _NT
```